Translating Data Between Geographic Information Systems

A thesis

submitted in partial fulfillment

of the requirements for the Degree of

Master of Science in Computer Science

in the

University of Canterbury

by

Richard T. Pascoe

University of Canterbury

1989

Table of Contents

Abstract

Transferring data from one geographic information system (GIS) to another is difficult because of the diverse, and often complex, structure of transfer file formats. Accordingly, the design and implementation of an interface for transferring data from one format to another is time consuming and difficult. The translation may be performed by an interface constructed for the two formats (the individual interfacing strategy), by two interfaces through an interchange format (the interchange format interfacing strategy), or by a number of interfaces through a series of formats (the ring interfacing strategy).

The interchange format interfacing strategy is widely adopted because it offers an acceptable compromise between the quality of the data translation and number of interfaces required. In contrast, the individual interfacing strategy achieves the best quality of translation but is generally rejected because of the impracticality of constructing a large number of interfaces.

The goal pursued in this thesis is to maximise the quality of the translation by overcoming the impracticality of the individual interfacing strategy. This is achieved in the following way. An interface is divided into three phases: the decode phase, in which the source format decoder places data from the source format into a relational data base; the translate phase, in which the data is restructured according to a translation algorithm written in a relational query language; and the encode phase, in which the target format encoder places data from the relational data base into the target format.

The time and effort involved in implementing these phases of data translation is minimised with the assistance of the following software tools: parser generators and lexical analysers which are used for generating format decoders; a relational data base management system which is used for implementing translation algorithms; and an encoder generator which is used for generating format encoders. The encoder generator is a new tool developed in this thesis. The efficacy of these tools is demonstrated, and a significant reduction in the effort of constructing interfaces is achieved, making the individual interfacing strategy a practical approach.

Chapter One

Introduction

Many organisations require the same geographic data. Organisations responsible for supplying electricity, telecommunications, and drainage, for example, have a common need for a digital representation of data such as coastal outlines, roads, and house boundaries. Data acquisition is achieved most frequently through the laborious procedure of hand digitising existing maps and editing this digital representation to achieve the desired quality of data.

It is wasteful for many organisations to capture the same data in this way; rather, the data should be digitised once and then made available to any organisation requiring it. In doing so, the enormous effort in capturing data by digitisation is performed once for all organisations. Furthermore, designating one source for shared data sets will result in a more consistent collection of data across organisations, and making this data readily available will reduce the time, effort, and cost of installing a new Geographic Information System (GIS).

Exchanging data is complicated because organisations use different geographic information systems, and these systems represent geographic data in different ways. For example, Colourmap, GeoVision, and GDS are three geographic information systems that have individual external data representations, or *transfer file formats*. A transfer file format (TFF) defines the structure of a set of files that may act as the import/export gateway for data that is being transferred in or out of a GIS; data exported from the system will be made available, and data to be imported into the system must be provided, in this format. To achieve the transfer of data from one type of GIS to another, the data must be translated from the transfer file format of the source GIS into the format of the target GIS.

This translation from one format to another is performed by a *geographic interface* which has been defined as "a mechanism by which one data structure can be directly converted into another data structure for the purpose of communication between systems or sub-systems" [van Roessel et al 1986]. The data structures referred to in the definition are taken to be transfer file formats.

Some formats are not associated with a specific GIS. Instead, these intermediate or *interchange formats* are used in conjunction with a particular type of *interfacing strategy* that defines the way in which data is exchanged between geographic information systems. Two examples of an interfacing strategy are: the *individual interfacing strategy,* where all GI Systems exchange data between each other directly; and the *interchange format interfacing strategy*, where all GI systems exchange data between each other indirectly through an interchange format. Interfacing strategies such as these have evolved in an attempt to reduce the number of interfaces necessary for translating data from one format into another.

The theme of this thesis is to reduce the effort of implementing a geographic interface to such an extent that the more desirable individual interfacing strategy can be applied. Employing this strategy allows the advantage of providing the optimum translation from one format to another to be gained. This theme is developed as follows.

In Chapter 2, a description is presented of the underlying data models that are the basis for the majority of the formats. A format is divided into two parts: a geographic data model, and a method of encoding this model into the transfer media. The geographic data model is further subdivided into a spatial model, and a descriptor data model. A notation is presented, and used throughout this thesis, for describing the various stages of the data translation process. A discussion is given of the goals to be achieved when considering translating geographic data.

In Chapter 3, a comparison is made of three interfacing strategies: the individual interfacing strategy; the ring interfacing strategy; and the interchange format interfacing strategy. Although the last of these strategies is widely adopted, it is shown by the author that the effectiveness of this strategy is reduced by the definition of many different interchange formats. In showing this weakness, the importance of being able to minimise the effort necessary for implementing an interface, regardless of the interfacing strategy adopted, is emphasized.

In Chapter 4, a description is given of schemes for translating data in the fields of electronic publishing, database exchange, and computer languages, as well as geography. The purpose of examining these schemes is to look for solutions to problems that are similar in nature to those to be found in translating geographic data. Particularly useful solutions have been incorporated into the author's method of designing, specifying or implementing a translation.

In Chapter 5, a structure is presented of a translation process for geographic data. The structure consists of three phases: the *decode phase*, the *translate phase*, and the *encode phase*. An explanation is given of how this structure serves the purpose of this thesis.

In Chapter 6, the specification of a data translation is described with the intention of processing this specification to implement the desired translation. In particular, a discussion is presented on the use of: various diagramming techniques to specify the geographic data model of a format; a BNF notation to specify the syntactic structure of files in a format; an extended relational query language to specify the translation algorithm.

Implementation is discussed of the decode, translate, and encode phases in Chapters 7, 8, and 9, respectively. These discussions centre on the use of software tools such as Yacc and Lex for constructing format decoders, and the use of relational data base management systems, such as INGRES, for storing and manipulating geographic data.

Chapter Two

Geographic Data Translation

The process of geographic data translation has many aspects that must be understood before a successful translation can be achieved. These aspects are: the different representations of geographic features used in various transfer file formats; the interfacing strategies that may be used; the steps needed to accomplish a translation; and the problems in implementing each of these steps. Each of these aspects is examined in detail in the following sections.

2.1. Geographic Data Representation

Geographic data consists of features: a feature is "a defined entity and its object representation" [DCDSTF 1988], with an entity being "a real world phenomenon that is not subdivided into phenomena of the same kind", and an object being "a digital representation of all or part of an entity".

Data associated with real world entities is divided into two categories: the spatial data, which "portray the spatial locations and configurations of individual entities" [Peuquet 1984] , and the non-spatial, attribute, or descriptor data which describe the non-spatial characteristics of the entities. For example, consider an object representing an oil well. The spatial data may describe the object as a point at some latitude and longitude. The descriptor data may define that point as an oil well with a name and rate of production.

Accordingly, the objects that represent the entities are defined by a *geographic data model* that is composed of: a model for spatial data, which defines the topological structures and geometry of the objects; and a model for descriptor data, which defines the attributes of, and the relationships between, the objects. A *transfer file format* consists of a geographic data model that is encoded by some *implementation method.* The implementation method is "a method of encoding data content and data structure to accomplish a transfer between dissimilar computer systems, without loss of content, meaning, or structure." [DCDSTF 1988].

Topology deals with the spatial configuration of an object, and conveys information about its spatial relationship, such as incidence and adjacency, with other objects. In the recently published Spatial Data Transfer Specification [DCDSTF 1988] the following set of cartographic objects are defined: 0-dimensional objects, points and nodes; 1-dimensional objects, lines, line segments, strings, arcs, links, directed links, chains, and rings; 2-dimensional objects: areas, interior areas, polygons, pixels, and grid cells.

Geometry deals with the location, size, shape, and orientation, of the objects within some coordinate system. The geometry is frequently specified using one of the following coordinate systems: Geographical (longitude and latitude), Universal Transverse Mercator, Lambert and, in New Zealand, the New Zealand Map Grid. Geometric data such as the area of a polygon, length of a line, and the shortest path between two nodes of a graph, are computed using the locations of objects within the coordinate system.

Because, in both their topology and geometry, spatial data models can be complex [van Roessel *et al* 1986, Peuquet 1984] there is a wide variety of spatial models. These models, however, derive from either the vector model, or the tessellation model [Peuquet 1984].

A connected sequence of x,y coordinates is the basic logical unit of a vector data model and it is used to construct more complex spatial objects such as polygon boundaries, or networks. Points are regarded as a special case where there is only one set of coordinates in the sequence. For example, in the GeoVision GINA transfer file format [GeoVision 1986] the linear feature type, commonly defined by a sequence of coordinates, also "includes single-point features". Peuquet points out that the following are all examples of the vector model: the spaghetti model, the topologic model, the GBF/DIME model, and the POLYVRT model.

The basic logical unit of a regular tessellation model is a pixel, or a grid cell. Peuquet notes that:

> *"....tessellation, or polygonal mesh models, represent the logical dual of the vector approach. Individual entities become the basic data units for which spatial information is explicitly recorded in vector models. With tessellation models, on the other hand, the basic data unit is a unit of space for which entity information is explicitly recorded."*

Peuquet identifies five forms of the tessellation model:

1) Grid and other regular tessellations. These are based around the three differing geometries of the square, triangular, and hexagonal meshes.

2) Nested tessallation models. These are based on subdividing the elemental polygon of the grid into polygons of the same shape. The most recognised of these models is the quadtree [Samet 1984].

3) Irregular tessellations. These differ from grid and other regular tessellation models in that the elemental polygons within a mesh are not necessarily of the same size. The most commonly used model of this form is called the triangulated irregular network (TIN).

4) Scan-line models. These are a special case of the square mesh where the cells of a mesh are organised into single, contiguous rows across the data surface, usually parallel to the x axis.

5) Peano scans. These scans are based on mappings of n-dimensional space onto lines and vice versa. Peano scans are found to be useful for image processing applications [Stevens *et al*, 1983]

For real world entities represented by objects in the geographic data model, a descriptor data model defines the attributes of, and the relationships between, those entities. Research into descriptor data models has, to a large extent, received less attention than research into spatial data models.

The problem of particular interest in this thesis is that of translating geographic data represented by a *vector spatial data* model and some form of a *descriptor data* model.

2.2. Notation of Data Translation

Geographic data translation can be described as follows: Data represented in some source transfer file format F_S is to be translated into some target transfer file format F_T, that is:

$$F_S \rightarrow F_T$$

When the interface is implemented, the data may be translated through a series of temporary formats, denoted by f_i, that are internal to the interface. These temporary formats are of interest only to someone wanting a detailed understanding of the interface, perhaps with a view to modifying it for use in a different translation. The notation emphasises this limited interest by using lower-case f inside braces, that is

$$F_S \rightarrow \{f_I \rightarrow f_J\} \rightarrow F_T$$

The translation process may also involve one or more interchange transfer file formats (Chapter 3), denoted by I. For example, a translation involving interchange transfer file format j can be described as:

$$F_S \rightarrow I_J \rightarrow F_T$$

The source to interchange format translation is performed by an *export interface*; correspondingly the interchange to target format translation is performed by an *import interface.*

Consider now the use of the notation developed above to describe a translation process to convert data provided by the Department of Survey Land and Information (DOSLI) for display on a system owned by the Christchurch Drainage Board. Source data was in the GeoVision GINA transfer file format [GeoVision 1986], and the target format was the BIF transfer file format of the GDS system [GDS 1984]. The translation from GINA format to BIF format was made through the interchange transfer file format SIF [Intergraph 1986]; that is:

$$F_{GINA} \rightarrow I_{SIF} \rightarrow F_{BIF}$$

This translation was achieved using the import interface $I_{SIF} \rightarrow F_{BIF}$ that was already available on the Christchurch Drainage Boards system, and the export interface $F_{GINA} \rightarrow I_{SIF}$ which was implemented by the author to demonstrate the techniques described in Chapters 7, 8 and 9 of this thesis. The export interface consisted of three phases and two temporary formats:

$$F_{GINA} \rightarrow \{f_{GRM} \rightarrow f_{SRM}\} \rightarrow I_{SIF}$$

In Chapter 5, an explanation is given of these phases and the temporary formats.

2.3. The Goals of Data Translation

Deciding on the priority of the goals when developing a translation process depends on the user's perspective. The geographic database administrator will expect the translation to produce data of the highest possible quality; designers who have to construct a large number of different translations wish to minimise the effort required to develop each translation; users who frequently employ interfaces will expect the interface to operate efficiently without placing an excessively high demand on computing resources.

Any data translation is achieved, therefore, with a compromise between the following conflicting goals:

1) To minimise the time and effort involved in constructing interfaces.

2) To maximise the quality of the translation.

3) To maximise the performance of the interface.

It is not possible to achieve all three goals simultaneously. The complexity of transfer file formats, and the goal of achieving a translation of high quality, results in a complex interface; accordingly, the time and effort necessary to implement the interface, and the amount of computing resources used by the interface while translating the data, will increase.

The goal of minimising the time and effort involved in the construction of an interface may be pursued by adopting methods to automate the construction of individual interfaces. The automatic generation of many types of programs has been a long term goal of computer science and three types of automatic programming systems have been described by Rich and Waters [1988]:

1) *Bottom up*. Programs are constructed using very high level languages which consist of powerful abstract data types and operations.

2) *Narrow Domain*. Programs are automatically generated by a program generator that is constructed for a specific type of program domain.

3) *Assistant*. Programs are constructed using powerful software tools such as: intelligent editors, on-line documentation aids, and program analyzers.

The goal of maximising the quality of the translation is approached by developing improved methods of specifying and implementing the mapping of the source data into the target format. In doing so, issues that determine the quality of translation may be addressed.

Examples of these issues are:

1) Can all that is represented in the source format be represented in the target format ?

2) Can cartographic objects from the source format be mapped onto equivalent cartographic objects in the target format ?

3) Can all data required in the target format be derived from that contained in the source format ?

4) If there are alternative representations in the target format, which is most suitable for the data being translated ?

For example, consider the translation

$$F_{BIF} \rightarrow F_{SIF}$$

In the BI Format, a circle is represented by its centre point, and radius. In the SI Format, a circle is represented by its centre point and radius, with the optional specification of a transformation matrix, and whether the circle is a solid, or a hole. It can be seen, therefore, that an accurate translation can be achieved from the BI Format representation of a circle to the SI Format representation. If the translation had of been

$$F_{BIF} \rightarrow F_{GINA}$$

the translation would have been almost impossible because there is no representation for a circle in the GINA format (some other representation such as a hexagon may be an acceptable substitution). Other examples can be given where a target format cannot represent data in some other format.

An important factor in the quality of data translation is the mapping of cartographic data objects from the source format onto objects of the target format. If the source system is designed for engineering or draughting, and the target is a geographic information system, then the mapping between the two may be difficult, as in the previous example involving the representation of a circle. The BI Format and the SI Format deal with precisely defined geometric shapes found in drawings, whereas the GINA Format deals with less precise geometric representations of natural phenomena.

In many cases, the target format may explicitly represent data that is implicit within the source format. For example, in the Colourmap format [CSIRONET 1986] the number of lines that define the boundary of a polygon is explicitly stored for each polygon in the polygon partition of a geographic map file. In the BI Format, however, this information is implicitly represented by the occurrence of the line primitive definitions associated with an object, and has to be calculated (see § 8.2, Example 1) for inclusion in the Colourmap format.

In some cases, data necessary for the target format is absent from the source format. For example, in the Colourmap format there is no record that specifies the coordinate system used for the data. This information is contained within the definition of the format, therefore, it has to be provided by whoever implements the translation. The coordinate system in use for the data within a GINA format is explicitly stated by the coordinate system record of the database header file.

An interchange format may have alternative representations for data. For example, the SDTS offers three representations, or transfer forms: the Vector form, the Relational form, and the Raster form. In some cases, these representations may be incompatible with those of the target format, consequently, either the translation cannot be done, or there is a loss of data quality because of the mapping from one representation to another.

The goal of maximising the performance of an interface is considered is probably of much less importance than the other two because [Penny 1986]:

> *...the conversion is a once-only operation for data that may be used hundreds of times. Simplicity of the transfer, rather than its efficiency, is the key factor .*

The emphasis in this thesis is, therefore, on the goals of reducing the time and effort taken to construct interfaces, and of achieving a high quality data translation.

Chapter Three

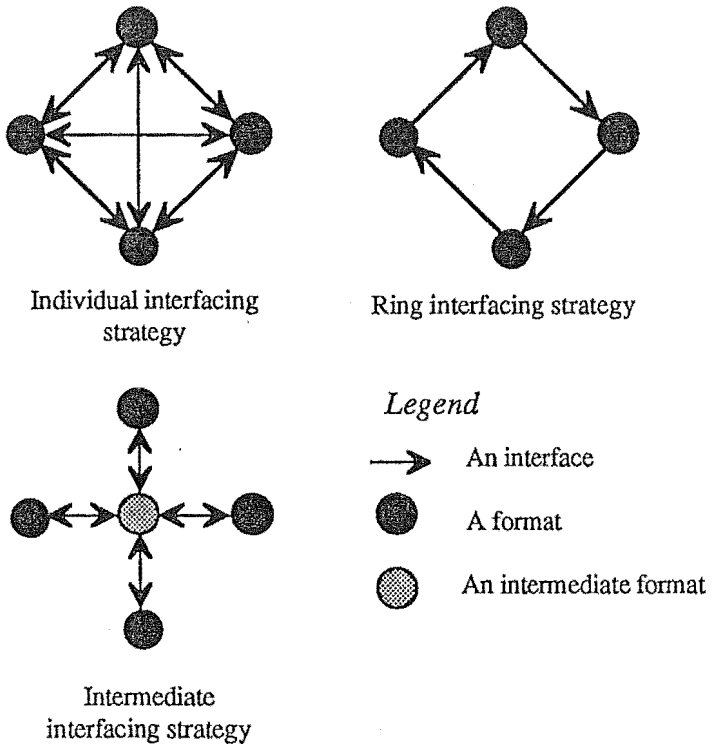
Interfacing Strategies

In an attempt to reduce the effort required to allow geographic information systems to communicate with each other, a number of different *interfacing strategies* can be considered. In general, an interfacing strategy specifies the way in which interfaces are organised to transfer data between geographic information systems. Three types of interfacing strategies (Figure 3.1) are described by Fosnight and van Roessel [1985]:

1) The *individual* interfacing strategy
2) The *ring* interfacing strategy
3) The *interchange format* interfacing strategy

Figure 3.1

Interfacing Strategies



The individual interfacing strategy employs an interface for each source-to-target format translation. Data in the source format is translated directly into the target format. The translation process is described as:

$$F_S \rightarrow F_T$$

The *ring interfacing* strategy organises the use of interfaces in a way that connects all formats in series with the the last format in the series connected to the first. The best situation is when the target format is the next in the series after the source format, in which case the process is described as above. Otherwise, data in the source format is translated into the target format through one or more intervening formats; that is:

$$F_S \rightarrow F_{I1} \rightarrow .... \rightarrow F_{Ik} \rightarrow F_T$$

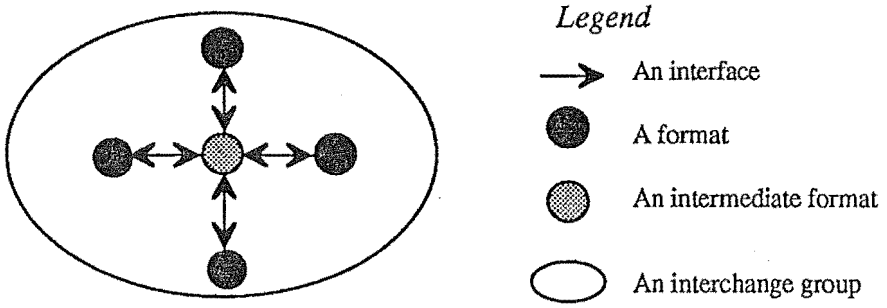
The *interchange format* strategy revolves around the use of one interchange format. Data in the source format is translated into an interchange format by the export interface, and then into the target format by the import interface. The translation process is described as:

$$F_S \rightarrow I_1 \rightarrow F_T$$

All those geographic information systems that have a pair of import and export interfaces for a particular interchange format will be referred to here as an interchange group (Figure 3.2). For example, all geographic information systems that have import and export interfaces for the SIF interchange format belong to one interchange group, and all those with import and export interfaces for the SDTS will belong to another.

Figure 3.2

An Interchange Group



3.1. Comparison of Interfacing Strategies

To compare the three strategies, suppose that there are N geographic information systems that exchange data with each other. Figure 3.3 presents the number of interfaces, *T*, that would have to be constructed for each of the three strategies. If an interfacing strategy were to be selected on the basis of minimising the total number of interfaces constructed, then the ring interfacing strategy would be the best. This basis is unsatisfactory for deciding on an interfacing strategy because it ignores the number of interfaces required to perform a translation for any pair of source and target formats.

Figure 3.3

The number of interfaces required according to each interfacing strategy

Interfacing strategy	*Total number of interfaces*
Individual Interfaces	T = N(N-1)
Ring Interfacing	T = N
Interchange format	T = 2N

There are two reasons for reducing the number of interfaces involved in any source-to-target format translation: fewer interfaces mean a quicker translation; and, more important, fewer intervening formats make it less likely that information will be lost. Information present in the source format and representable in the target format, may be lost because the intervening formats are unable to represent that information. For each interfacing strategy, Figure 3.4 presents the for any pair of source and target formats.

Figure 3.4

The number of interfaces required to perform a translation from one format to another

Interfacing strategy	*Number of interfaces in a translation*
Individual Interfaces	1
Ring Interfacing	up to N-1
Interchange format	2

The number of interfaces required for a translation when operating under the ring interface strategy may be very high, up to N-1. The potential loss of information, therefore, makes the strategy undesirable. With the individual interface strategy, no information representable in both the source and target formats need be lost. If the interchange format strategy is used, no information representable in both the source and target formats will be lost, *provided* that the interchange format can represent data that may be represented in any format of that interchange group.

Use of the interchange format strategy is wide-spread because an acceptable compromise is reached between the total number of interfaces constructed, and the number of interfaces used in a data translation. The use of the interchange format strategy does leave some problems unresolved and these are examined in the next section.

3.2. Interchange Formats

Many interchange formats have been defined [Penny 1986]. During research for this thesis, the author has become familiar with three: the Standard Interchange Format (SIF) [Intergraph 1986], the Spatial Data Transfer Specification (SDTS) [DCDSTF 1988], and the proposed New Zealand Transfer File Format (NZTFF) [van Berkel 1987].

Without specifying the rules governing how many interface groups a format may belong to, the number of interfaces, T, which guarantees that data in one format, can be translated to and from any other, is given by the following formula:

$$T = 2NM$$

where there are N formats and M interchange formats.

If an interfacing strategy is introduced between the different interfacing groups, each format need only belong to one interchange group. Depending on which interfacing strategy is used, the number of interfaces, T, can be given by any of the following formulae:

1) the individual interfacing strategy,

$$T = 2N + M(M-1)$$

2) the ring interfacing strategy,

$$T = 2N + M$$

3) The interchange format interfacing strategy,

$$T = 2N + 2M$$

With, or without an interfacing strategy, the definition of many interchange formats has resulted in an increase in the number of interfaces necessary to guarantee that data in one format can be translated to and from any other. In short, the initial interchange format interfacing strategy is undermined by the definition of many interchange formats.

Another problem with an interchange format is the consequences of modifying its definition. It is conceivable, and to be expected, that the definition of any interchange format will alter over a period of time. A new representation for geographic data may be developed, or changes to current representations used within the interchange format may be required. For example, another transfer form of the SDTS may be specified for a hybrid data model such as the vaster data model proposed by Peuquet [1984]. When the SDTS was published, the US Geological Survey was made "the designated maintenance organisation" responsible for any revisions or modifications to the SDTS.

Once an interchange format becomes established and a pair of error-free import and export interfaces is implemented for each format operating under the interfacing strategy, data may be transferred between any of the formats. If, after a period of time, the interchange standard were to change possibly all interfaces would have to be modified. Emphasis should, therefore, be placed on designing interfaces in a way that facilitates their redesign after any possible change to the interchange, or transfer file format of a geographic information system.

Chapter Four

General Data Translation

The problem of data translation occurs in many areas other than geographic information systems, and many of the techniques developed in these areas may be adapted for geographic data translation. In the following sections, a description is presented of the techniques used to achieve language translation, electronic manuscript translation, and database translation. The Chapter ends with a description of research into geographic data translation by van Roessel *et al* at the EROS data center.

4.1. Language Translation

The problem of translating geographic data from one format to another is analogous to the problem of translating a computer program from one language to another. The translation of a program is performed by a compiler which translates from a high level language, say L_{HLL}, into a machine code, denoted L_{MC}. The notation of §2.2 described geographic data translation from a source format, F_S to a target format F_T, as:

$$F_S \rightarrow F_T$$

Similarly, the translation of a program can be described as:

$$L_{HLL} \rightarrow L_{MC}$$

Steel [1960] introduced the idea of an intermediate language, denoted here as L_I. Incorporating this into the description of the translation of a program results in the following:

$$L_{HLL} \rightarrow L_I \rightarrow L_{MC} \quad .$$

The translation is divided into two parts: the *front end* of a compilation translates the program written in the target language into the functionally equivalent program written in an intermediate language; the *back end* of a compilation translates a program written in the intermediate language into a functionally equivalent program in machine code. An example of this type of translating environment is that provided by the Amsterdam Compiler Kit [Tanenbaum *et al* 1983] which uses the EM intermediate language.

Suppose it is necessary to translate any one of N higher level languages into any one of M machine code languages. Figure 4.1 presents a comparison on the number of translators (compilers or front ends and back ends) necessary with, and without, the use of an intermediate language. It is apparent from Figure 4.1 that the use of an intermediate language reduces the programing effort from N*M compilers down to N front ends and M back ends. Furthermore, if another higher level language is developed then with the use of an intermediate language only one front for the new language would have to be implemented because all of the existing back ends can be used to complete the translations. Without the intermediate language, however, another M compilers, one for each of the available machine code languages, would have to be developed.

Figure 4.1

Translation method	*Number of translators*
Without intermediate language	N * M compilers
With an intermediate language	N front ends + M back ends

The drawback to the use of an intermediate language is that it is not possible to design a single intermediate language that provides a suitably small range of instructions for every combination of high level language and machine code language. The EM intermediate language [Tanenbuam 1978] is designed for use with block structured high level languages such as Algol and Pascal.

In the area of geographic data translation, the use of an intermediate format is analogous to the use of an intermediate language. The front end, and back end of a compiler, are equivalent to the export interface and the import interface, respectively. The same drawback that occurs with intermediate languages also occurs with intermediate formats. To avoid restricting the translation of data from the source format to the target format, an intermediate format must provide a wide range of data representations. The SDT Specification illustrates this by incorporating three transfer forms to cater for a wide a range of source and target format combinations.

The construction of compilers, including front ends and back ends, has been extensively studied, and a number of software tools have been developed to assist in this construction. In particular, the use of parser generators, such as Yacc and its companion lexical analyser Lex, is widely established in compiler construction.

Parser generators generate programs based upon the grammar of the high level language. The author has applied a parser generator to the construction of a class of programs, called *format decoders*, that decode the implementation method of the source format and place the data into a temporary data structure. The way in which this type of program is used, and implemented, is described in Chapter 7.

4.2. Electronic Manuscript Translation

The exchange of electronic manuscripts is a problem similar to that of exchanging geographic data. In an article describing the Chameleon research project at the Ohio State University, Mamrak *et al* [1987] makes the point that: "the wide variety of electronic-manuscript representations presents an obstacle to widespread exchange". The same problem applies for the exchange of geographic data.

In an attempt to reduce the problem of different representations for electronic manuscripts, many standard forms have been introduced. It is recognised, however, that standard forms "only reduce the number of translations required", and that "there are many different standards being proposed within the domain of electronic manuscripts, both nationally and internationally...Thus the support of translation among standard forms themselves may become necessary." [*op cit*]. There is an obvious similarity between this problem and that created by the occurrence of many intermediate formats for geographic data. The formation of interchange groups due to the many different intermediate formats for geographic data was discussed in § 3.1.

The primary goal of the Chameleon research project is to "develop a software system that will (1) support programmers in building software tools to do translations, and (2) provide assistance in the use of these tools while translating manuscripts into and from standard-form representations." In short, the objectives are to design and implement a comprehensive translation architecture that supports both the building and use of translation tools.

The Standard Generalized Markup Language [ISO 1986] is used to specify the translation from a standard to a nonstandard form. The software for electronic manuscript translation is constructed by processing this specification, using a set of tools that:

1) produce an attribute grammar that formally specifies the translation,

2) inverts the formal translation grammar of 1) to produce the translation from a nonstandard to a standard form,

3) generates scanners for the standard and nonstandard forms,

4) generates the software to implement the translation from a standard to a nonstandard form, and from a nonstandard to a standard form.

The method taken to the automatic generation of software for electronic manuscript translation is an example of the *assistant approach,* described in §2.3. This approach has been taken by Mamrak *et al* because, in both the construction and the use of electronic manuscript translation software, there are aspects which are not readily automated. Mamrak *et al* found that, "the intent of the author.....cannot be derived automatically". Consequently, experienced users are required to apply effectively the software tools that assist in the construction of the translation software, and to oversee the use of translation software. As explained in Chapter 5, the author has taken the assistant approach to the construction of software for geographic data translation.

4.3. Database Translation

In the area of database translation, the *bottom up* approach (§2.3) to the generation of data translation software has been researched in the development of the Extraction, Processing, and Restructuring System EXPRESS [Shu *et al,* 1977]. EXPRESS was "originally developed as a research prototype in order to test the generality and applicability of applying high-level data description and high-level data manipulation techniques to the data translation problem." [Taylor 1982]. For specifying the desired translation, the system provides two nonprocedural languages: DEFINE for data description, and CONVERT for data manipulation.

The DEFINE language is used to specify the, usually hierarchical, structure of data files. According to this specification, a program is generated that either reads data from the source files, or writes data into the target files. As well as describing the structure of files, the language provides facilities for: editing data, checking of data integrity, and incorporating user defined error detection and correction procedures.

The CONVERT language is used to manipulate hierarchically structured data that, for convenience, may be viewed as a *form.* Figure 4.2 is an example of a form taken from Shu *et al* [1977]. In a relational data model, a form would correspond to an unnormalised relation.

Figure 4.2

(Department)								
DNO	MGR	Budget	(EMP)		(Proj)			
			ENO	Job	PJNO	Leader	(Equip)	
							Item No	Description
D1	DOE	40000	19	ENG	J6	RAE	221	Computer
			41	SEC	J8	MEW	46	Scope
			52	TECH			317	Laser
			77	ENG	J11	FAR	271	Computer
D4	SO	20000	60	CHEM	J9	LA	47	Microscope

The CONVERT language provides nine high level operators for manipulating the data. To illustrate the language, a brief description of three of these operations is provided here. The SLICE operator is used to transform hierarchical data into a flat file. For example, the following SLICE operation on the form in Figure 4.2 produces the form presented in Figure 4.3(a):

```
A = SLICE (DNO, MGR, PJNO, LEADER, DESC, FROM DEPT);
```

The SELECT operation selects part of a form which satisfies specified criteria. For example, the following applied to Figure 4.2 would produce the form shown in Figure 4.3(b):

```
B = SELECT(DNO, MGR, PROJ(PJNO, LEADER) FROM DEPT WHERE DESC EQ 'COMPUTER');
```

Figure 4.3

A				
DNO	MGR	PJNO	Leader	Description
D1	DOE	J6	RAE	Computer
D1	DOE	J8	MEW	Scope
D1	DOE	J8	MEW	Laser
D1	DOE	J11	FAR	Computer
D4	SO	J9	LA	Microscope

Figure 4.3(a)

B			
DNO	MGR	(Proj)	
		PJNO	Leader
D1	DOE	J6	RAE
		J11	FAR

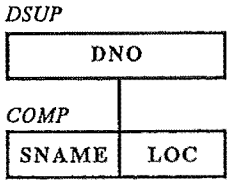
Figure 4.3(b)

The GRAFT operation is used to attach data to a hierarchical tree. For example (based on one from Shu *et al* [1977]), the form shown in Figure 4.4(a) is grafted onto (b) to create the form shown in Figure 4.4(c) by the following operation:

```
T = GRAFT (DSUP ONTO T BEFORE PJNO WHERE T.DNO = DSUP.DNO);
```

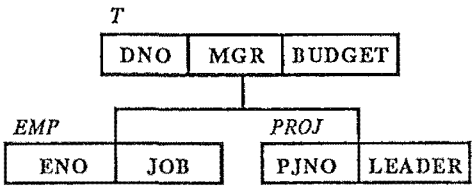
Figure 4.4

An illustration of a GRAFT operation



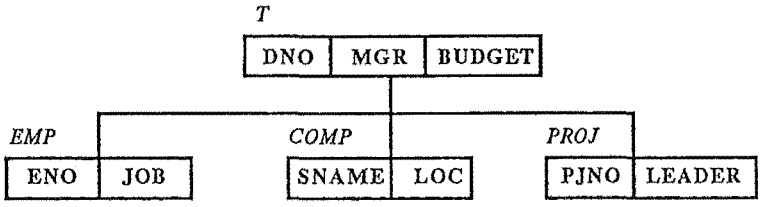
DSUP		
DNO	*COMP*	
	SNAME	LOC
D1	ACME	SF
	EMCO	MV
D5	DELTA	LA
	OXA	NY

Figure 4.4(a)



T						
DNO	MGR	BUDGET	*EMP*		*PROJ*	
			ENO	JOB	PJNO	LEADER
D1	DOE	40000	19	ENG	J6	RAE
			41	SEC	J8	MEE
			52	TECH		
			77	ENG	J11	FAR

Figure 4.4(b)



T								
DNO	MGR	BUDGET	*EMP*		*COMP*		*PROJ*	
			ENO	JOB	SNAME	LOC	PJNO	LEADER
D1	DOE	40000	19	ENG	ACME	SF	J6	RAE
			41	SEC	EMCO	MV	J8	MEE
			52	TECH				
			77	ENG			J11	FAR

Figure 4.4(c)

From the specification of the translation using the languages DEFINE and CONVERT, EXPRESS generates a set of PL/1 programs which, collectively, achieve the desired translation. The translation process consists of three major steps: the *read step* where the data in the source format is read, checked for errors, and organised into an internal form; the *restructuring step* where the data is structured into the target files; and the *load step* where the target files are loaded into the target database.

The idea of manipulating data using high level operators is sound, and the idea of using *relational* operators available with relational DBM Systems has been discussed [Penny 1986, van Roessel and Fosnight 1985]. The author stores and manipulates geographic data within a relational database management system called INGRES [Held *et al* 1975, Date 1986].

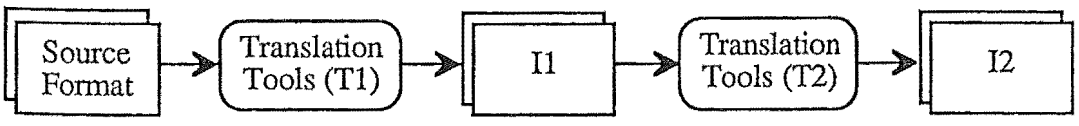
4.4. Geographic Data Translation

The EROS Data Center has been working on a project called the Spatial Data Research and Support System (SDRSS) with a goal to providing "an integrated set of system resources to support data acquisition, storage, processing, analysis, and product generation requirements of a broad research program directed toward the integration and application of disparate spatial data types". During the development of SDRSS, research has been performed into the design and implementation of geographic interfaces to enable the transfer of vector data between geographic processing systems [van Roessel *et al*, 1986].

The I2 intermediate data structure developed by van Roessel *et al* is an implementation of the intermediate structure used in the interfacing model proposed by the Working Group on Data Organisation of the National Committee for Digital Cartographic Data Standards (NCDCDS) [Nyerges, 1984]. The proposed model (Figure 4.5) is the same as the interchange format interfacing strategy described in § 3. The model, however, only describes the translation from the source to the intermediate format; the translation from the intermediate format to the target transfer file format is similar.

Figure 4.5

The interfacing model proposed by the Working Group on Data Organisation of the National Committee for Digital Cartographic Data Standards.



According to the NCDCDS workgroup model, there are two categories of tools, T1 and T2. Tools in category T1 are used to translate data into the "intermediate" intermediate (sic) data structure I1. Tools in category T2 are used to translate data from the I1 structure into the intermediate structure, I2.

If data is to be exchanged by employing the interfacing model proposed by the NCDCDS, two interfaces are required: the export interface, to translate from the source GIS format into the intermediate format, and the import interface, to translate data from the intermediate format into the target GIS format. When supplying data, the source system provides the data in the standard interchange format using the export interface. The data is then translated by the import interface of the target system into the native transfer file format.

van Roessel *et al* have implemented this interfacing model, using the relational data model for three reasons:

1) elegance and simplicity of the data representation

2) the availability of the relational algebra and its unique relational operators

3) the availability of a number of different software systems on different hardware configurations

The I2 intermediate data structure is defined as an interchange format, and the data is maintained in a relational database management system; the RIM relational database management system was used at the EROS Data Center. The I2 data structure consists of six core relations containing spatial data:

regpol:	region number, polygon number
polarc:	polygon number, arc number
archdr:	arc number, start node, end node, left region, right region
arcxy:	arc number, x, y
nodearc:	node number, arc number
nodexy:	node number, x, y

and a set of nested relations containing primary and secondary non-spatial data:

attprime:	element number, attribute 1, attribute 2, attribute 3, ...
attsec:	attribute 1, secondary attribute 1, secondary attribute 2,....

The thrust of the research by van Roessel *et al* was to minimise the time taken to construct an interface, through the application of the relational operators in the translation process. In particular, the operators were considered useful as T2 tools. Tools in category T1, such as the program RIMNET developed at the EROS Data Center, are used to translate data into the I1 data structure. The function of RIMNET is to transform an unnormalised data representation into a normalised representation. The transformation is specified using a free-format syntax that defines the desired tracking sequence of data elements through the unnormalised form.

Van Roessel *et al* also introduced the idea of using Backus Naur Form (BNF) to describe the syntactic structure of a format. A discussion on this method of specifying the structure of a format is deferred until Chapter 6. The approach taken in this thesis, which is described in the next Chapter, improves on that of van Roessel *et al*. by extending the use of BNF, and using it to generate software that extracts data from the source format and place it into a relational data base.

Chapter Five

The Translation Process

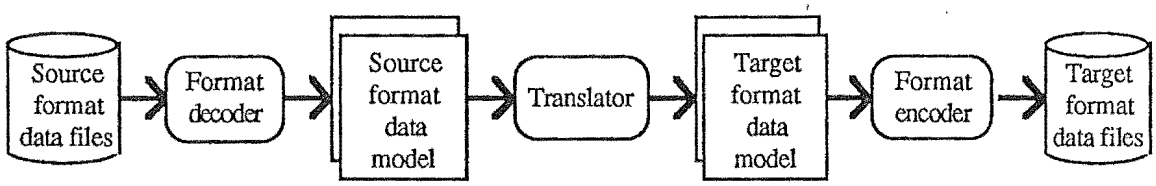
The translation process is defined here as the method used to restructure data from the source format into the target format. The author's model of the translation process consists of three phases (Figure 5.1):

1) the *decode* phase, which extracts data from the source transfer file, and places it into an equivalent relational data model

2) the *translate* phase, which reorganises data from the source relational data model into the target relational data model

3) the *encode* phase, which retrieves data from the relational data model and encodes it for the target transfer file.

Figure 5.1



The translation of data from the GeoVision GINA format to the SIF format has been described (§2.2) as

$$F_{GINA} \rightarrow \{f_{GRM} \rightarrow f_{SRM}\} \rightarrow I_{SIF}$$

In the decode phase, the data is transferred from the GINA format F_{GINA} into this format's relational data model f_{GRM}:

$$F_{GINA} \rightarrow f_{GRM}$$

In the translate phase, the data is mapped from the GINA relational data model f_{GRM} into the SIF relational data model f_{SRM}:

$$f_{GRM} \rightarrow f_{SRM}$$

In the encode phase, the SIF relational data model is encoded into the SIF interchange format I_{SIF}:

$$f_{SRM} \rightarrow I_{SIF}$$

Each phase is to be performed by the user of the translation system. The user, most likely a technician, would have to be familiar with at least geographic data structures and computer storage media such as tape and floppy disk.

Three software tools are used in the translation process, each in one particular phase of this process. In Chapter 7, the use of parser generators and lexical analysers is described for implementing the decode phase of the translation process. In Chapter 8, the use of a relational DBMS is described for storing and manipulating the data into the target data model. In Chapter 9, a description is given of a new software tool, designed by the author, for implementing the encode phase of this process.

To use these tools effectively, the technicians who translate the data will have to develop expertise in using relational database management systems, and parser generators. Experience gained with EXPRESS (§4.3) revealed the problem of achieving a balance between developing a translation system that is sufficiently general to be useful, and developing one that is easy to use [Taylor, 1982]. In the Chameleon Project (§4.2), it is acknowledged by Mamrak *et al* [1987] that only users who are "experts in the theory and practice of formal languages" can effectively use the translation system which was developed. To gain the maximum benefit from translation systems that require experienced users, Taylor [1982] suggested that "a designated center of conversion expertise" could be set up.

The rationale behind this three phase translation process is primarily to develop software modules that can be used for many different translations, thereby reducing the effort of implementing any future interfaces. For example, if the GINA to SIF translation

$$F_{GINA} \rightarrow \{f_{GRM} \rightarrow f_{SRM}\} \rightarrow I_{SIF}$$

had been implemented and, later, data in the Colourmap format was to be translated into the SIF interchange format, that is

$$F_{CMP} \rightarrow \{f_{CRM} \rightarrow f_{SRM}\} \rightarrow I_{SIF}$$

then only the decode phase, $F_{CMP} \rightarrow f_{CRM}$, and the translate phase, $f_{CRM} \rightarrow f_{SRM}$, would have to be implemented because the encoder phase, $f_{SRM} \rightarrow I_{SIF}$, used in the GINA to SIF translation could be used again. For any further translation from the Colourmap format, the decode phase, $F_{CMP} \rightarrow f_{CRM}$, could be used.

Other advantages of the approach taken are that the modular architecture will minimise the impact of modifications to a format, and the division of the translation process into phases reduces the task into smaller problems that can be solved with the assistance of software tools. It can be classified, therefore, as an *assistant approach* to the automatic generation of interfaces.

In Chapter 3, the advantages and disadvantages were discussed of three different interfacing strategies: the individual interfacing strategy, the ring interfacing strategy, and the interchange format interfacing strategy. The translation process described here can be used to provide an interface for use in any of these three interfacing strategies. The individual interfacing strategy, however, is favoured by the author because:

1) the effort required to implement the greater number of interfaces required for that strategy is reduced through the repeated use of the encode and decode phase implementations for each format and the use of relational database management systems in the translate phase of the translation process

2) the use of the individual interfacing strategy increases the quality of the translation for the reasons described in Chapter 3.

The next Chapter describes how to specify a particular source to target format translation in a form that facilitates the implementation of each of the three phases of the translation process.

Chapter Six

Translation Specification

According to van Roessel *et al* [1986], "one of the first steps for developing a conversion methodology is to obtain a consistent description" of a translation process. In this Chapter the form is given of the specification for a translation process which is to be implemented using the approach described in Chapter 5.

The description of a translation process should be in a form that is suitable for its intended use. To automate the implementation of a translation process, much of the specification should be in a form that can be used by software tools which are employed to implement the process.

Corresponding to the three phases of the translation process described in Chapter 5, there are three parts to specifying a particular translation process:

1) A description of the source format

2) A description of the steps to be taken when translating data form the source format to the target format

3) A description of the target format.

In section 6.1, the way in which the description of a source format, used in the *decode* phase, and the description of the target format, used in the *encode* phase, is discussed. The specification of the source to target format translation, used in the *translation* phase, is discused in section 6.2.

6.1 Format Specification

In section 2.1, a transfer file format was defined as consisting of a geographic data model encoded into files according to some implementation method. In the definition of the SDTS [DCDSTF 1988], for example, the geographic data model is referred to as the conceptual data model and the ISO 8211 [1985] (the equivalent British standard is BS 6690 [1986]), entitled "A data descriptive file for information interchange", is specified as the implementation method.

To encode and decode a format, specification is required of the geographic data model and the implementation method. The next two sub-sections describe the way in which the data model and the implementation method are to be specified when using the method of translating geographic data suggested here.

6.1.1 Geographic Data Model Specification

The geographic data model of a format defines the topological structures and geometry of the objects, and the attributes of, and the relationships between, these objects. The definition of a geographic data model is used in the design of a relational structure for storing the data from a format. Two techniques for specifying the geographic data model of a format will be discussed here: dependency diagrams as used by Smith [1985], and entity-relationship diagrams [Martin and McClure, 1986].

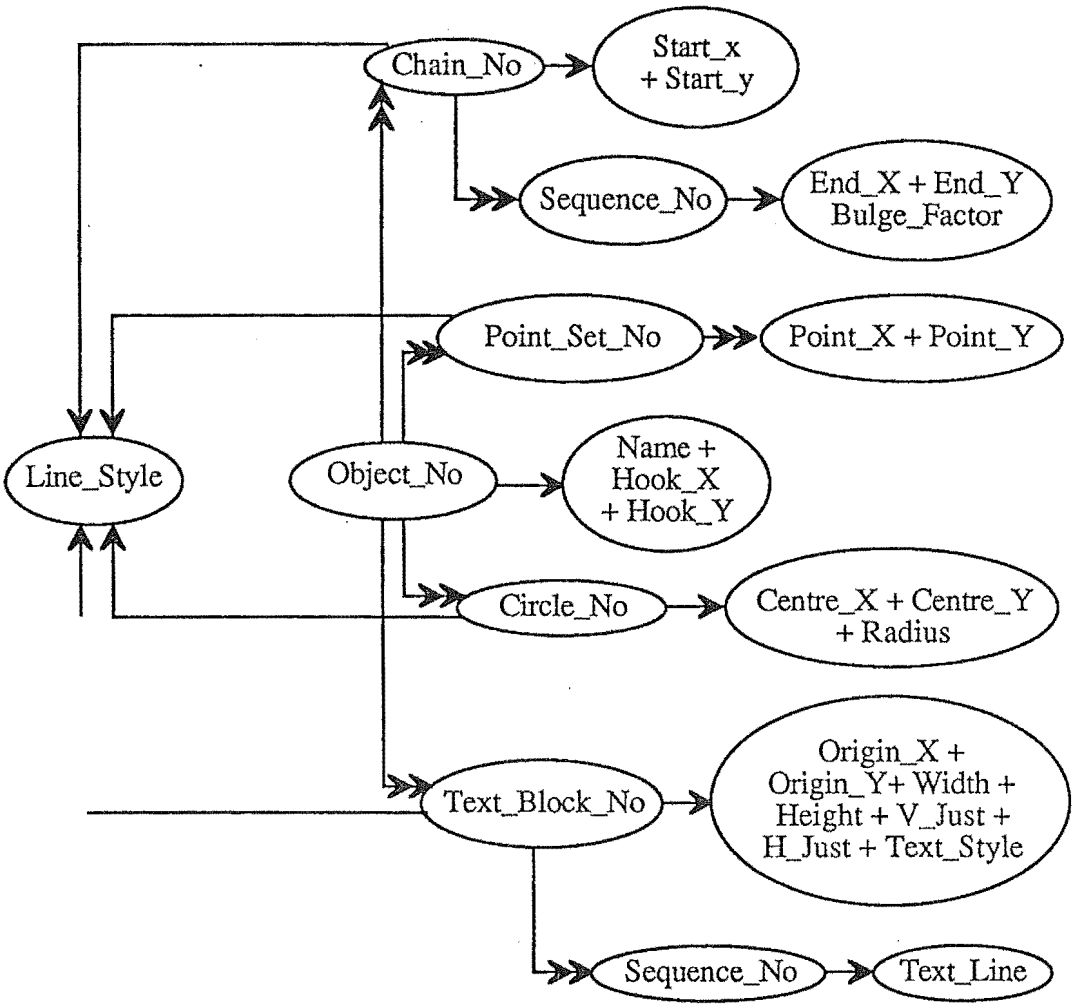
Smith [1985] defines a procedure that enables fully normalized relations to be directly composed from a completed dependency list and dependency diagram. The list and diagram are used to specify the single-valued and multivalued dependencies between data fields. For a single-valued dependency to exist from data field A to data field B, one fact about A must determine a single fact about B. Each value of A must be nonnull and unique and any value of B may be null or duplicated. For a multivalued dependency to exist from A to B , one fact about A must determine a *set* of facts about B.

Each data field of the data base is pictorially represented in a dependency diagram by enclosing the field's name within an ellipse. There may be more than one field within an ellipse. These ellipses are interconnected by lines with one or two arrow heads depending on whether they represent single-valued, or multivalued dependencies. A set of rules for the construction of a dependency diagram is given in a paper by van Roessel [1987] that describes the application of Smith's method to the design of a spatial data structure for a relational data base. In Figure 6.1, a dependency diagram is presented for the character form of the Binary Interface File format.

Entity-relationship diagrams provide an alternative technique for specifying the geographic data model of a format. In the context of entity-relationship diagrams, an entity is something "(real or abstract) about which we store data" [Martin and McClure, 1986]. An entity type is "a named class of entities which have the same set of attribute types"[*op cit*]. An attribute of an entity type describes a property of an entity. For example the entity type Chain may have the attributes Chain_id, Left_polygon_id, Right_polygon_id, Start_node_id, and End_node_id.

Figure 6.1

Dependency diagram for the character form of the BIF format geographic data model



In an entity-relationship diagram, an entity type is represented by a named box and the attributes of an entity type are represented by an ellipse containing the attribute names. The entity type is connected to its attributes by a line and the associations between different entity types are represented by links. There are four types of associations [Martin and McClure, 1986] :

1) *Basic associations,* which indicate how many instances of one entity can be associated with another entity,

2) *Labeled associations,* which are basic associations with text that describe what the association represents,

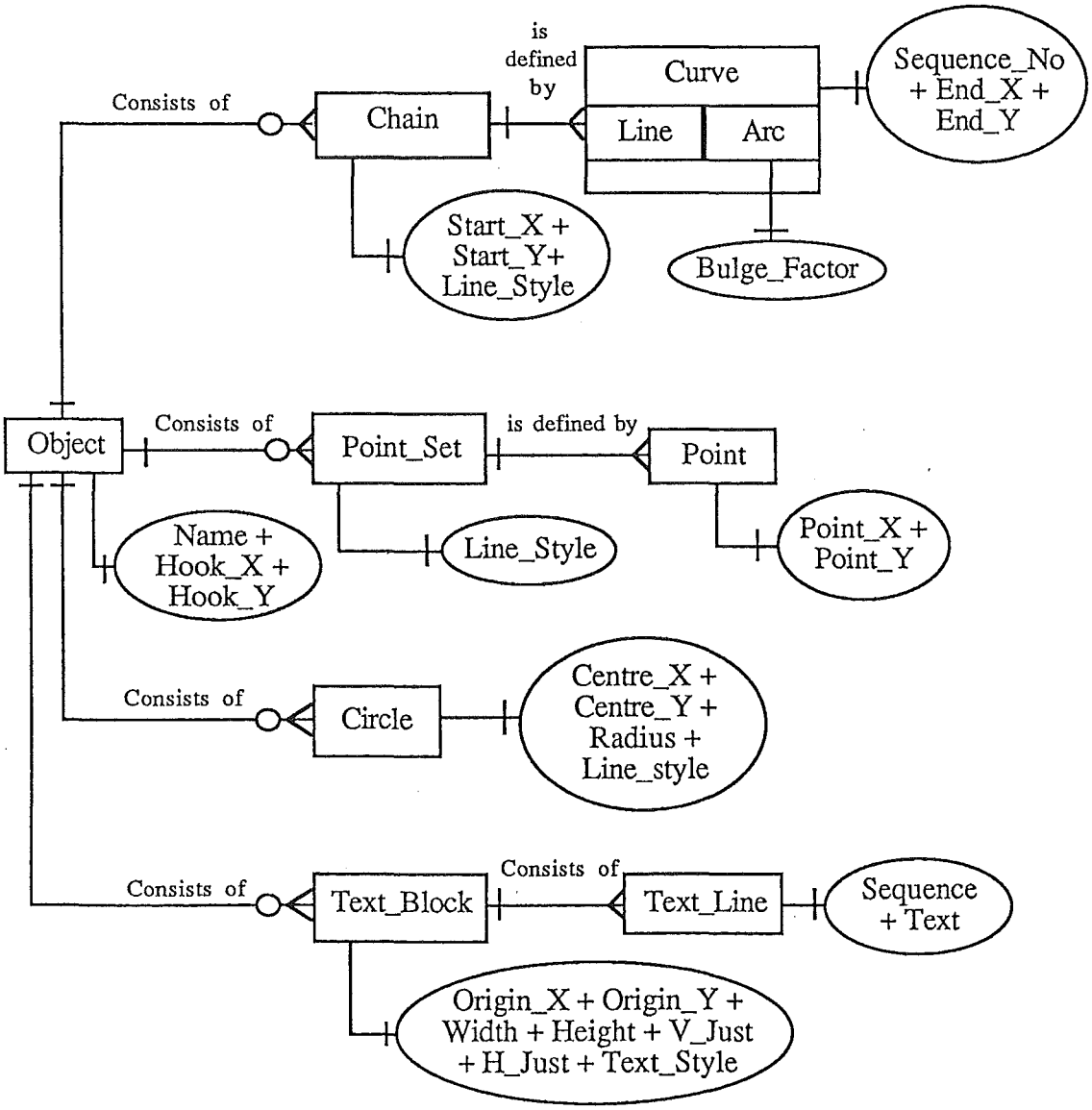
3) *Looped associations,* which indicate basic, or labeled associations within the same entity type,

4) *Linked associations,* which indicate that there is some type of connection between the associations made; for example, associations may be linked together to indicate that all must exist for an occurrence of the entity types involved.

Three types of notation are used for entity-relationship diagrams: Crow's-foot notation, Arrow notation, and Bachman notation. In Figure 6.2, an entity-relationship diagram for the geographic data model of the character form of the BIF format is presented using Crow's-foot notation.

Figure 6.2

An Entity-Relationship diagram for the character BIF geographic data model



To compare the two techniques for specifying a geographic data model, consider the specification of the model for the character form of the Binary Interface File format using a dependency diagram (Figure 6.1) and an entity-relationship diagram (Figure 6.2).

In an Entity-Relationship diagram, the individual entities such as objects, chains, and circles, are explicitly represented whereas in a dependency diagram, entities are implicitly represented by the occurrence of the attributes associated with these entities. Because of this implicit representation in the dependency diagram, it can be difficult to specify structures such as the line and arc entity subclasses shown in the entity-relationship diagram of Figure 6.2. Another advantage of using entity-relationship diagrams is that associations, such as linked associations, between entities can be specified.

In conclusion, either a dependency or an entity-relationship diagramming technique can be used to document the geographic data model of a format. Entity-relationship diagrams are preferred by the author because they can be used to provide more detail on the relationships between entities represented in a format.

6.1.2 Implementation Method Specification

The specification of a format's implementation method will define at least the following:

1) the file structure, that is, what records occur within a file and the order in which these records occur,

2) the record structure, that is, what fields occur in a record, whether this occurrence is compulsory or optional, and whether the field repeatedly occurs in the record,

3) the field structure, that is, what type of values are found in each field.

Van Roessel *et al* [1986] suggested the use of a notation based on Backus Naur Form (BNF) as a method for concisely specifying the implementation method of a format. BNF, as used by Naur [1963] to define the syntax of ALGOL 60, consisted of metalinguistic formulae such as

<unsigned integer> ::= <digit> | <unsigned integer> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Characters enclosed in the brackets < > represent metalinguistic variables whose values are sequences of symbols. A metalinguistic formula consists of two parts: the left hand side, which specifies the variable being defined; and the right hand side, which specifies the valid sequences of symbols either directly, by listing the symbols themselves, or indirectly, through the use of variables. The two parts of a formula are separated by the connective symbol ::=, and alternative sequences on the right hand side of a formula are separated by the connective symbol |. The order in which a symbol, or a variable occurs in a formula indicates the order in which they occur in the sequence of symbols being defined.

In addition to the ::= and | metalinguistic connectives used by Naur [*op cit*], van Roessel *et al* [1986] used an ampersand to indicate that both metalinguistic variables either side of the ampersand occur in the sequence, although not necessarily in the specified order. Curly brackets are used by van Roessel *et al* to indicate that the enclosed variables may be repeated an unspecified number of times and square brackets are used to indicate that the enclosed variables are optional.

In this thesis, BNF as defined by Naur [*op cit*] is augmented by the use of a *lexicon* for defining those metalinguistic variables that consist of a sequence of symbols. An entry in a lexicon consists of: a token name, which is used in the BNF specification where the sequence of symbols would be expected to occur; and the regular expression that defines the sequence of symbols that the token name refers to. In Figure 6.3, a BNF specification and a lexicon is given for the implementation method of the character form of the Binary Interface File format, and the specification of the implementation method for other formats are given in Appendix A.

A regular expression is a method of defining a regular set that consists of character sequences. In general, a regular expression is defined over a character set using operators that indicate repetition, and alternatives. There are various notations, and operators for defining regular expressions; the following examples are defined according to the requirements of the lexical analyser lex[Lesk and Schmidt 1979]:

[0-9]+ denotes all sequences of digits of length 1

(ab|cd) denotes either the character sequence *ab*, or the sequence *cd*

(ab?c) denotes either the character sequence *ac*, or the sequence *abc*

(ab|cd+)?(ef)* denotes character sequences such as *abefef*, *efef*, *cdef*, or *cddd*

Figure 6.3

Specification of the Implementation method of the character form of the BIF format

«datafile» ::= «empty» | «drawing» «object list»

«empty» ::= *(that is, the null string of symbols)*

«drawing» ::=DRAWING REAL REAL REAL REAL

«object list» ::= «empty» | «object list» «object»

«object» := «name» «hook» «primitive list»

«name» ::= NAME «ocd value»

«ocd value» ::= «compound value» | «ocd value» COLON «compound value»

«compound value» ::= «a value» | «compound value» «a value»

«a value» ::= INTEGER | REAL | STRING

«hook» ::= HOOK REAL REAL

«primitive list» ::= «primitive» | «primitive list» «primitive»

«primitive» ::= «line» «segment list» | «circle» | «text» «text line list» | «points» «point list»

«line» ::= LINE REAL REAL «compound value»

«segment list» ::= «to» | «arc» | «segment list» «to» | «segment list» «arc»

«to» ::= TO REAL REAL

«arc» ::= ARC REAL REAL REAL

«circle» ::= CIRCLE REAL REAL REAL STRING

«text» ::= TEXT REAL REAL REAL REAL REAL REAL STRING «a value»

«text line list» ::= «chars» | «text line list» «chars»

«chars» ::= CHARS STRING

«points» ::= POINT STRING

«point list» ::= «at» | «point list» «at»

«at» ::= AT REAL REAL

Figure 6.3(a): the BNF specification

Lexical Symbol	*Regular Expression*
ARC	"arc"
AT	"at"
CHARS	"chars"
CIRCLE	"circle"
COLON	":"
DRAWING	"drawing"
HOOK	"hook"
INTEGER	[+-]?[0-9]+
LINE	"Line"
NAME	"Name"

Lexical Symbol	*Regular Expression*
NEWLINE	"\n"
POINT	"point"
REAL	([+-]?[0-9]+"."[0-9]* ([E][+-][0-9][0-9])*) | ([+-]?[0-9]*"."[0-9]+ ([E][+-][0-9][0-9])*)
STRING	([\\,\-A-Za-z'$#!%^& ~?/]+[0-9]*)+
TEXT	"text"
TO	"to"

Figure 6.3(b): the lexicon

BNF has become an established method for describing the syntax of computer languages after it was used to describe the syntax of ALGOL 60 [Naur 1963]. Consequently, software tools for processing these BNF descriptions have been developed for use in the construction of compilers. In Chapter 7, a description is presented of how the software tools Yacc and Lex, primarily intended to be used in the construction of parsers, are used to implement a format decoder for the decode phase.

6.2 Specifying the Source to Target Format Mapping

A *translation algorithm* defines a procedure for reorganising data from the source format into the target format. This procedure may involve:

1) reorganising data from the source format into the structure of the target format

2) deriving data not contained explicitly within the source format

The use of high-level operators for concisely defining this procedure has been found to be successful [Taylor 1982, van Roessel *et al* [1986], Penny 1986]. As suggested by van Roessel *et al* (see § 4.4), and Penny [1986], a translation is specified here using the structures and operators of the relational data model defined by Codd [1970].

6.2.1 The Relational Data Model

At the core of the relational data model are relations, and a set of operations for manipulating these relations. Relations are two dimensional tables where a column of a table is referred to as an attribute, and the rows of a table are referred to as tuples. Codd [1972] originally defined eight operators, which can be divided into two groups [Date 1986]: the traditional set operations union, intersection, difference, and cartesian product where the interpretation of these have been modified for use with relations; and the special relational operations select, project, join, and divide.

Expressions for manipulating relations using these operators can be formulated in two functionally equivalent ways: relational algebra, and relational calculus. A solution formulated using relational algebra explicitly specifies a sequence of relational operators that will construct the desired relation. Using relational calculus, the necessary operations to construct the desired relation is derived from a description of this relation.

To illustrate the difference between relational algebra and relational calculus, a data base consisting of the following relations is defined: the polygon relation , consisting of the attributes polygon_name, and ring_name; and the ring relation, consisting of the attributes ring_name, sequence_number, and chain_name. Consider the formulation of an expression for constructing the relation poly_chain, which contains the set of chains that define the boundary of a polygon called "POLY2". Using relational algebra, the relational expression would be:

> Join relations polygon and ring on the attribute ring_name;
>
> From the result of that join, select tuples whose polygon_name attribute has a value of "POLY2";
>
> From those tuples selected, project on the attributes polygon_name, and chain_name

Using relational calculus, the relational expression would be:

> Get polygon_name and chain_name for polygons such that there exists a polygon and a ring with the same ring_name value and the polygon_name attribute has a value of "POLY2"

The INGRES relational DBM System provides a query language QUEL, which is an implementation of relational calculus [Date 1986]. The use of this language for specifying a translation algorithm is discussed in Chapter 8.

In an article describing the use of a relational query language for specifying a "conversion algorithm" to reorganise data from the source format into the target format, Penny [1986] concluded that a set of spatial operators, in addition to the relational operators, would have to be provided by a query language. Systems that provide, or are designed to facilitate the provision of, such operators are described next.

6.2.2 Extending the Relational Model

As a consequence of applying the relational data model to the storage and manipulation of geographic data, new data base management systems are being designed and implemented with this application in mind. Two examples of this evolution are: POSTGRES [Stonebraker and Rowe 1986], and a model built around Geo-Relational Algebra [Guting 1988].

POSTGRES (POST inGRES) is a new relational data base management system designed and implemented to incorporate new functions that are difficult to integrate with the existing data base management system INGRES. Of particular interest here are the design goals of:

1) supporting complex objects such as polygons, lines, and circles,

2) masking it easier to extend the data base management system to provide: specialized data types such as a latitude and longitude position data type for mapping applications; and new operators for manipulating these data types.

Guting [1988] has proposed a model and query language for geometric data base systems based on relational algebra. The new language, geo-relational algebra, introduces the following into relational algebra:

1) attributes of relations may have geometric data values such as points, line, or regions, and new operators such as inside, outside, intersects, is_neighbour_of, and perimeter, are defined to manipulate these data values,

2) a tuple of a relation describes an object as a combination of geometric and non-geometric attribute values, and a relation consists of a homogeneous collection of geometric objects.

These advances in relational data base technology provide an ideal environment for reorganising geographic data from one format into another. The specification of this reorganisation is given as an algorithm written using a relational query language that has been extended to include spatial operators. In Chapter 8, a discussion is presented on specifying these translation algorithms using the INGRES relational calculus query language QUEL.

Chapter Seven

The Decode Phase

The purpose of the decode phase of a translation process is to decode the data from the implementation method of the format, and to place this data into a relational data base.

Implementation methods for encoding the geographic data model of a format may be divided into those methods that encode the model in a binary file representation, and those that encode the model in a text file representation. In a binary file representation the smallest unit of storage in a file is the bit whereas in the text file representation the smallest unit is a character from a character set such as the American Standard Code for Information Interchange (ASCII).

The primary advantage of a binary file representation is that less space is required to represent data than that of a text file representation. The disadvantage is that the binary representation of integer and real values vary for different computers, and purpose built software must be used to access the files. A text file representation is that it can be stored on any machine that uses the same character set as that of the representation.

The method of implementing the decode phase that is discussed in the following sections is applicable for implementation methods that use text file representations. For data encoded in a binary file representation a different approach to implementing the decode phase would have to be taken. For example, in the case of the Binary Interface File format (BIF) a utility for converting the binary representation into a text representation was used by the author. The resultant text representation was processed by software that was constructed using the method described in the following sections.

In section 7.1 a description is given of how the software tools Yacc [Johnson 1979] and Lex [Lesk and Schmidt 1979] are used to generate a format decoder. The interface between the format decoder and the relational data base management system INGRES is discussed in § 7.2 where a set of functions for simplifying the interface is described. Techniques to solve specific problems raised when implementing decoder modules are given in § 7.3. Finally, in § 7.4 a few comments are made on the importance of efficient memory management when dealing with very large volumes of data.

7.1. Parser Generators

In Chapter 4 (§ 4.1), an analogy between geographic data translation and language translation was made. The function of a parser in a compiler is analogous to that of a format decoder: a parser recreates the structure of the source program in the form of a parse tree; a format decoder recreates the structure of the source format in a relational data base. This similarity is exploited to implement a format decoder using the same software tools that are used to generate a parser.

A parser generator, of which Yacc is an example, is a software tool for generating the parser of a compiler according to a BNF description of the language to be parsed. A lexical analyser such as Lex is used to generate a routine for breaking a sequence of characters into tokens according to regular expressions and passing these tokens onto the parser. These tools are used to generate a format decoder by processing the BNF and lexical components of a format specification (§ 6.1).

Although the Unix parser generating tools Yacc (Yet Another Compiler Compiler) and Lex are used here, any software which is functionally equivalent to Yacc and Lex can be substituted for these tools.

Grammars that are to be processed by Yacc use a notation which differs from that used by Naur (§ 6.1.2) in the following ways:

1) A different connective symbol is used for separating the meta-linguistic variable from its definition; the symbol ":" is used instead of the symbol "::="

2) A *token name* is substituted for the sequence of characters that define the metalinguistic variable; consequently, the variables are not enclosed by the symbols "<" and ">" because there is no distinction to be made between those symbols that form the variable names, and those symbols that occur in the data being parsed.

The token names used in the grammar are defined by regular expressions (see § 6.1.2), and these definitions are processed by Lex to generate the lexical analyser. When a sequence of characters is matched with a regular expression, the lexical analyser signals to the parser that the token associated with the expression has been read.

Actions to be performed by the parser are associated with formulae in a Yacc grammar. These actions are written using the C programming language [Kernighan and Richie 1978] and are enclosed by the symbols "{" and "}". When using Yacc to generate a parser, these actions usually construct the parse tree. When using Yacc to generate a format decoder, these actions place the data into a relational data base. A set of routines for accessing data in a data base that is managed by INGRES is described in § 7.3.

To illustrate the use of the tools Yacc and Lex, the syntactic specification and associated actions are given in Figure 7.1 for the header record of an attribute file in the Colourmap format. In Appendix B the Yacc and Lex definition files are given for the Colourmap and BIF format decoders.

Figure 7.1

Syntactic specification and associated actions for the header record of an attribute file in the Colourmap format

```
header        : num.attr num.names missing.dat.val NEWLINE
     {
        eqlappend("cm_attrdss","nattributes",itv($1),"nnames",
              itv($2), "missdatval", dtv($3), EOAL);
     };

_num.attr_    : INTEGER
     { $$ = $1; };

_num.names_   : INTEGER
     { $$ = $1; };

_missing.dat.val_ : REAL
     { $$ = $1; };
```

Lexical Symbol	*Regular Expression*
INTEGER	[+-]?[0-9]+
NEWLINE	"\n"
REAL	([+-]?[0-9]+"."[0-9]* ([E][+-][0-9][0-9])*) | ([+-]?[0-9]*"."[0-9]+ ([E][+-][0-9][0-9])*)

In Figure 7.1, the header record is defined to consist of three data elements: `num.attr`, an integer value which specifies the number of attributes in the file; `num.names`, an integer value which specifies the number of names, or objects that are included in the file; and `missing.dat.val`, a real value used to indicate that there is no attribute value available for this combination of name and attribute. When this record is found, the action performed by the decoder is to call the C function, `eqlappend()`, that is provided for appending data to a relation.

In § 7.3, a discussion is presented on two techniques for solving problems encountered while constructing format decoders for the GINA format, the Colourmap format, and the text representation of the BIF format. A complete listing is given of the definition files for these format decoders in Appendix D. In the next sub-section, a description is given of a set of function calls for placing data into a data base managed by the relational data base system INGRES.

7.2. Simplified INGRES Interface

The relational data base INGRES provides an embedded query language, called EQUEL, which allows customised user interfaces to be constructed using a combination of the general programming language C and the QUEL query language of INGRES. A preprocessor translates the EQUEL statements embedded in an application into library calls. The application is then compiled, and the INGRES libraries are linked in to resolve these library calls.

To illustrate EQUEL syntax, the following example appends the value "Christchurch" to the `map` attribute of the relation `mapname` in the data base called `example1`:

```
main()
{
##      ingres example1
##      append mapname(map = "Christchurch")
##      exit
}
```

As the format decoder processes the source format, it either appends data to relations, or replaces existing data in relations. To remove the necessity of using the pre-processor, and to provide some independence between the decoder and the source relational data model, the author developed two functions, eqlappend(), and eqlreplace(). These functions construct parametrized QUEL statements at run-time by retrieving from the INGRES data base a description of the attributes involved in the query. Thus, if the size of an attribute is changed, the decoder will continue to operate because the description of the attribute is not specified at compile time.

The following BNF grammar defines the argument structure for these two functions:

```
«eql append arg list»   :   «relation name» «element list»
                            «EOAL constant»

«eql replace arg list» :    «relation name» «element list»
                            «EOAL constant» «qualification»
                            «EOQL constant»

«element list»:             «attribute name» "," «attribute value»
                        |   «element list» "," «element list»
```

The following calls to these functions illustrate the use of these functions:

```
eqlappend( "relation", "attribute1", attr1_value, "attribute2",
           attr2_value, EOAL);

eqlreplace( "relation", "attribute2", new_attr2_value, EOAL,
           "relation.attribute1 = 200", EOQL);
```

A listing of these two functions, and others that are used to implement them is given in Appendix C.

7.3. Decoding Techniques

Two readily identified tasks of a format decoder are the introduction of sequence numbers (§ 7.3.1), and the introduction of default values where they are omitted in the format (§ 7.3.2). Techniques to fulfill these tasks are likely to be useful when implementing many format decoders; therefore, a description of these techniques are given in the following sub-sections.

7.3.1. Repeating Groups

A particular problem with transferring data from a sequential structure such as a file into a relational data base management system is that of processing *enumerated repeating groups* (ERG). An enumerated repeating group consists of a list of data objects preceded by the number of objects in the list. Within the Colourmap transfer file format, for example, part of the definition of a segment consists of an enumerated repeating group which specifies the connected sequence of points that form the segment.

To illustrate an ERG, consider the BNF specification of an enumerated list of coordinate pairs as follows:

```
«coordinate list»  ::=      «number of coordinates»
                            «coordinates»

«coordinates»      ::=      «coordinate pair»
                   |        «coordinates» «coordinate pair»

«coordinate pair»  ::=      «x» «y»
```

The relationship between the data element «number.of.coordinates» and the number of coordinate pairs in the list cannot be explicitly defined using a BNF grammar. Therefore, to incorporate this relationship into a format decoder generated by Yacc and Lex, the author has used the following technique.

Yacc permits an action to be performed before parsing of a production in the grammar is completed. The author uses this facility to set a counter to the number of data objects in the repeating group. The BNF grammar for the Colourmap transfer file format is augmented with an End Of Repeating Group (EORG) token. Using the notation required for processing by Yacc, the above illustration of an enumerated group of coordinate pairs is specified as follows:

```
coordinate.list:        number.of.coordinates
                        {
                            INITRG(number.of.coordinates);
                        }
                        coordinates EORG

coordinates:            coordinate.pair
                  |     coordinates coordinate.pair

coordinate.pair:  x y
                            { DECRG; }
```

Each time the lexical analyser is called to supply a token to the parser, the lexical analyser checks to see if a repeating group is being parsed and, if so, whether the last data object in the group has been found. If the last data object in the group has been recognised, then the lexical analyser generates an EORG token. Otherwise it continues reading the datafile and generating tokens in the usual manner. Each time the parser recognises a data object, the counter is decremented. Appendix D lists the data structures, macros, and alterations needed to the definition of the lexical analyser to implement this technique. Examples of its use can be found in the definition files for the Colourmap source decoder (Appendix B).

When decoding an enumerated repeating group and placing the data objects into the source relational data model, the format decoder may have to provide an explicit sequence number for each data object in the repeating group. The sequence number may be required to comply with the non-ordering property of the relational model as specified by Codd [1970].

Generating the sequence number for each data object in an enumerated group can be achieved through the use of the facility, provided by Yacc, for associating a value with a symbol in a grammar. This value is calculated by the actions defined for the alternative definitions for the symbol defined. Consider the ERG of coordinate pairs used above. To sequence explicitly the coordinate pairs, the following set of Yacc rules would be used:

```
1) coordinate.list:     number.of.coordinates
                             { INITRG(number.of.coordinates); }
                        coordinates
                        EORG

2) coordinates:         coordinate.pair
                             { $$ = 0; }
3)                  |   coordinates coordinate.pair
                             { $$ = $1 + 1 }

4) coordinate.pair:     x y
                             { DECRG; }
```

The non-terminal symbol `coordinates` is defined by rules 2 and 3. Associated with this symbol is a value which is the sequence number of the most recently seen coordinate pair. To ensure that the coordinate pairs are correctly sequenced in ascending order from left to right, rules 2 and 3 are "left recursive" grammar rules. Rule 2 will be reduced for the first coordinate pair only; therefore, the first coordinate pair will always have a sequence number of 0. The second and all succeeding coordinate pairs are taken care of by rule 3. The sequence number for rule 3 is calculated by incrementing the sequence number associated with the `coordinates` symbol.

7.3.2. Lexical Analysis of Datafiles

In the Colourmap transfer file format specification, some data elements are character strings. Because a character string may contain alphabetic characters, spaces, and digits without being enclosed by quotation marks, the length of a character string is the only way of ensuring that the correct number of characters are associated with a data element.

Character strings in the Colourmap format have different lengths depending on what the data element is: a comment is 80 characters long, a zone name is 10 characters long, and the site name or annotation for a point is 40 characters long. A technique for enabling the format decoder to instruct the lexical analyser on the length of the next character string (a STRING token) is as follows.

Yacc provides the facility of performing an action before a rule is completely recognised; therefore, the length of the required character string can be specified in front of the character string in a rule. For example, a rule for the zone identifier with a character string length of 10 is:

```
_zone.id_:          { datasize = 10;}
                        string
```

The lexical analyser uses the value assigned to the variable `datasize` to determine how long the expected string token will be. There is a complication with this technique: before the action is performed, the next token must have been returned by the lexical analyser, that is the string token would have to be read in before the action setting its length is performed.

To solve this problem, the rule

```
string:                 SETSIZE STRING
```

for character strings is defined. Then the token (SETSIZE) following the action which sets the length of the character string will always be generated by the lexical analyser before the STRING token with its associated character string is returned. In effect, the lexical analyser sends a message (the SETSIZE token) to the format decoder requesting more information before returning a STRING token. Incorporating the rule for character strings with the above example using the zone identifier would result in

```
_zone.id_:          { datasize = 10; }
                        SETSIZE STRING
```

When the lexical analyser recognises a sequence of characters that match the regular expression for a STRING token, it checks to see whether the size of the character string has been set by the parser. If it has been set, the lexical analyser takes the number of characters required leaving any excess characters for the next token. Otherwise, the lexical analyser leaves all of the matched characters for the next call and generates a SETSIZE token.

7.4. Processing Large Data Volumes

Because of the potentially large volumes of data being processed, the performance of the decoder can be adversely effected if insufficient care is taken towards memory management. From the outset, the goal of maximising the performance of software used in the translation process has been of secondary importance to the goals of minimising the construction time for interfaces, and maximising the quality of the translation (§ 2.3). The issue of efficient memory management was, however, considered of significance in the construction of a format decoder.

A decoder primarily uses memory as temporary storage locations for data that is being transferred from the source format into the relational data base. If this memory is not relinquished for future use, the large volume of data uses large amounts of virtual memory. As a result of this high demand on memory, the decoder takes longer to complete its task because it is suspended with increased frequency while its request for memory is processed. Experience with a format decoder processing a source format indicated that the memory required by the decoder was modest, and reached a stable level *provided* that memory was relinquished after having been used.

Chapter Eight

The Translate Phase

During the translate phase of the translation process, data from the source format is reorganised into the relational data model of the target format. As discussed in § 6.2, this reorganisation is to be specified by a translation algorithm written in an extended relational query language. In this thesis, the example algorithms are written in the INGRES query language QUEL, which is described in § 8.1.

In section 8.2, a discussion is presented on implementing the translation algorithm using QUEL and finally, in section 8.3, an explanation is given on how the choice of interfacing method is influenced by use of a relational query language to specify the translation algorithm.

8.1. QUEL

QUEL is a calculus-based language for data manipulation within data bases that are managed by INGRES. A brief description of this language follows, and a more detailed description can be found in the INGRES Reference Manual [Kalash *et al* 1986], and Date [1986]. A QUEL interaction consists of at least one range statement, and one or more command statements. A range statement of the form

RANGE OF variable-list IS relation-name

defines a set of tuple variables that range over a relation. These variables are used in command statements to identify tuples of relations. Command statements of the form

Command [result-name] (target-list) [WHERE Qualification]

can be grouped into: update commands, such as APPEND, DELETE, or REPLACE, which append or delete tuples in relations, or replace attribute values in tuples of a relation; and the RETRIEVE command, which is used to inquire about data contained within the data base.

To illustrate the difference between relational algebra and calculus, in § 6.2.1 the following example query was formulated:

> Get polygon_name and chain_name for polygons such that there exists a polygon and a ring with the same ring_name value and the polygon_name attribute has a value of "POLY2"

As an example of the RETRIEVE command, this query would be written in QUEL as:

```
RANGE OF  plygn IS polygon
RANGE OF  rng IS ring
RETRIEVE  poly_chn(polygon_name = plygn.polygon_name,
                   chain_name = rng.chain_name)
  WHERE   plygn.ring_name = rng.ring_name and plygn.polygon_name = "POLY2"
```

QUEL does not provide spatial operators such as those provided in the geo-relational algebra proposed by Gutting (see § 6.2.2). Instead of modifying the query language, a filter for providing these operators was implemented by the author. The filter intercepts operations of the form

> &operator(argument_list)

and maps them onto functions provided by the filter. For example, the operator

> &impbif(source_filename)

results in the execution of the BIF source decoder, which takes data from the named BIF file and places it into the data base.

8.2. Translation Algorithms

When implementing translation algorithms using a programming language such as C [Kernighan and Ritchie 1978], the specification of the mapping from the source format to the target format is intertwined with other statements for maintaining the data structures, or controlling the flow of program execution. Translation algorithms can be implemented more concisely, clearly, and with less effort, by using an augmented relational query language, such as the combination of QUEL and a filter, as described above for providing spatial operators.

To illustrate the use of QUEL for specifying translation algorithms, the following examples are taken from algorithms written by the author.

Example 1: calculating the number of objects that collectively define another object.

When performing the translation phase from the BIF format to the Colourmap format,

$$\{F_{BRM} \rightarrow F_{CRM}\}$$

the number of lines that define the boundary of a polygon has to be calculated for a polygon definition in the Colourmap format. This can be achieved by counting the number of graphic primitives associated with an object representing a polygon in the BIF format. QUEL provides the aggregate operator COUNT() for counting the number of tuples that meet a specific qualification and it can be used in the following way.

Consider a relational model of the BIF format that includes the relation obj_prm with attributes object_name, prm_name, and prm_type. Each tuple in this relation links the definition of a graphic primitive with an object definition. To create a relation obj_nmbr_prm that contains the attributes object_name, and nmbr_prm, the following QUEL query would be used:

```
RANGE OF objprm IS obj_prm
RETRIEVE obj_nmber_prm( object_name = objprm.object_name,
                        nmbr_prm = COUNT(objprm.prm_name by objprm.object_name))
```

Applying this query on the relation shown in Figure 8.1(a) results in the relation shown in Figure 8.1(b).

Figure 8.1

Calculating the number of line primitives that define a polygon in the BIF format

obj prm		
object name	*prm name*	*prm type*
A	1	line
A	2	line
B	3	line
B	4	line
B	5	line

Figure 8.1(a)

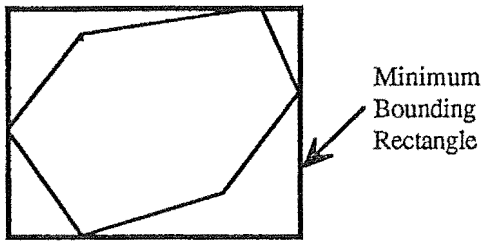
obj nmbr prm	
object name	*nmbr prm*
A	2
B	3

Figure 8.1(b)

Example 2: calculating the spatial extent of an object.

When translating from the GINA format to the Colourmap format, it is necessary, as part of the Colourmap format, to calculate the minimum bounding rectangle of each polygon (Figure 8.2). The minimum bounding rectangle delimits the spatial extent of an object and is specified by its diagonally opposing corners. The edges of the rectangle are parallel to the coordinate system axis.

Figure 8.2



The minimum bounding rectangle for an object is calculated by retrieving the minimum and maximum x, and y coordinate values of the object. For this example, the following relations are defined: poly_line, with attributes poly_name, line_name, direction, and edge; and feat_xy with attributes feat_name, feat_type, seq, x, and y; and the relation poly_mbr, with attributes poly_name, min_x, min_y, max_x, and max_y.

To construct the desired relation poly_mbr, the following sequence of QUEL statements is used:

```
RANGE OF pol_ln IS poly_line
RANGE OF feat IS feat_xy
RETRIEVE tmp(poly_name = pol_ln.poly_name, x = feat.x, y = feat.y)
   WHERE pol_ln.line_name = feat.feat_name and feat.feat_type = "line"
RANGE OF t IS tmp
RETRIEVE poly_mbr( poly_name = t.poly_name, min_x = MIN(t.x by t.poly_name),
        min_y = MIN(t.y by t.poly_name), max_x = MAX(t.x by t.poly_name),
        max_y = MAX(t.y by t.poly_name))
```

Applying the above sequence of QUEL statements to the relations shown in Figure 8.3(a) results in the relations presented in Figure 8.3(b).

Figure 8.3

Calculating the minimum bounding rectangle of objects in the BIF format

poly line			
poly name	*line name*	*direction*	*edge*
1	1	left	e
1	2	right	e
2	3	left	e
2	4	left	e
2	5	right	e

feat xy				
feat name	*type*	*seq*	*x*	*y*
1	line	0	1.4	2.3
1	line	1	2.2	3.4
1	line	2	3.1	1.3
2	line	0	3.1	1.3
2	line	1	1.4	2.3

Figure 8.3(a)

tmp		
poly name	*x*	*y*
1	1.4	2.3
1	2.2	3.4
1	3.1	1.3
1	3.1	1.3
1	1.4	2.3

poly mbr				
poly name	*min x*	*min y*	*max x*	*max y*
1	1.4	1.3	3.1	3.4

Figure 8.3(b)

Another aspect to the design of translation algorithms is that of coordinate transformations. Geometric data can be specified using a variety of projections and when transferring data from one GIS to another, it may be necessary to transform this data from one projection onto another. These projection transformations [Hutchinson 1981] are mathematically complex and not easily specified using a relational query language. The specification of projection transformations would be made easier by including in a query language a set of mathematical operators that assist in these transformations.

During the research for this thesis, the author has translated data from Christchurch Drainage Board (in the BIF format), into a relational data structure that is being used for the development of a map browsing facility [Penny *et al* 1989]. The algorithm used for this translation is given in Appendix E.

In summary, the author found that the specification of a translation algorithm to be easy using a query language such as QUEL because of the powerful operators, and the higher level of data abstraction, provided by the INGRES data base management system.

Chapter Nine

The Encode Phase

The purpose of a *format encoder* is to fetch data from the relational data base and encode it into the file structure of the desired target format. To automate the construction of interfaces, a new software tool is proposed that generates an encoder by processing a BNF specification of the file structure. This method of generating a program is similar to using Yacc to generate a format *de*coder (Chapter 7) and one specification of the file structure for a format should serve both tools.

In section 9.1 a discussion is presented on how the INGRES programing interface EQUEL influences the structure of encoders. In section 9.2 a prototype software tool for generating encoders is described and in section 9.3 the generation of an encoder for the character form of the BIF format is described.

9.1. EQUEL

A program written in the C programing language can retrieve information from a data base managed by INGRES through an embedded query language called EQUEL. EQUEL statements are QUEL statements (see § 8.1) preceded by the symbol "##". The facilities provided by EQUEL are identical to those of QUEL, except for the syntax and meaning of an EQUEL retrieve statement with no result relation specified. This form of EQUEL retrieve statement gives the C program access to the data retrieved. Its syntax is as follows [Kalash *et al* 1986]:

```
## retrieve (C-variable = a_fcn {, C-variable = a_fcn })
```

optionally followed by:

```
## [where qualification ]
## {
        /** C program statements **/
## }
```

The C program statements are executed for each tuple that is returned by the retrieve statement.

The use of such a retrieve statement is illustrated in Figure 9.1. The output shown in Figure 9.1(c) is generated by applying Figure 9.1(a) to the relation given in Figure 9.1(b).

Figure 9.1

Illustration of the EQUEL embedded query language

```
print_objectxy_relation()

/**
 **   variables object_id, x and y are previously defined and
 **   tuples are sorted by objectxy.objid and objectid.seq
 **/


{
## retrieve (object_id = objectxy.objid, x = objectxy.xcrd,
##           y = objectxy.ycrd)
## {
      printf("%s %lf %lf\n", object_id, x, y);
## }
}
```

Figure 9.1(a)

objectxy			
objid	seq	xcrd	ycrd
A	0	1.1	1.5
A	1	2.5	2.3
B	0	2.2	3.4
B	1	3.1	3.2
B	2	1.5	1.9

Figure 9.1(b)

```
A 1.1 1.5
A 2.5 2.3
B 2.2 3.4
B 3.1 3.2
B 1.5 1.9
```

Figure 9.1(c)

A parametrized retrieve statement is used when the structure of the retrieve statement is unknown at compile-time of an EQUEL program. The syntax of a parametrized retrieve statement for the commercially available INGRES is as follows [INGRES 1986]:

retrieve (**param**(*target string, varaddr*))

optionally followed by:

[where qualification]
{
    /** C program statements **/
}

A parametrized retrieve statement uses a function param() to process a target string and an array of pointers to variables. The target string is of the form

"%variable_type = a_fcn {, %variable_type = a_fcn}"

where the variable type may be one of the following:

i2 - two-byte integer
i4 - four-byte integer ("int" or "long")
f4 - four-byte floating point number ("float")
f8 - eight-byte floating point number ("double")
c - character string

When executing the parametrized retrieve statement, the INGRES run time system substitutes each variable type with its corresponding variable pointer taken from the varaddr array. Thus the sequence of pointers in the array, starting with varaddr[0], should correspond to the order of appearance of the variable types in the target string. The following fragment of code illustrates the form of a parametrized retrieve statement:

```
    /** variable declarations **/
##  char *tlist, char *varaddr[10];
##  int int_var;
##  double double_var;

    tlist = "%f8 = r.col1, %i2 = r.col2";
    varaddr[0] = &double_var;
    varaddr[1] = &int_var;
##  retrieve (param(tlist, varaddr))
##  {
        printf("%lf %d\n", double_var, int_var);
##  }
```

9.1.1. Nested Retrieve Statements

A retrieve statement is referred to as a nested retrieve statement when it occurs inside the body of another retrieve statement, as in the case of the italicised retrieve statement in Figure 9.2. The nested retrieve statement uses, as part of its qualification clause, data that is fetched by the outer retrieve statement. In Figure 9.2, data retrieved into the variable C-variable1 is subsequently used to restrict the tuples fetched by the nested retrieve statement.

Figure 9.2

An example of a nested retrieve statement

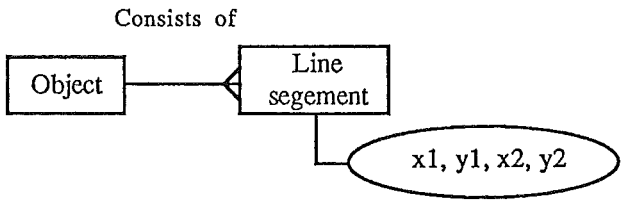
```
## retrieve (C-variable1 = rel1.attr1 , C-variable2 = rel1.attr2 )
## {
        /** C - program statements **/
##      retrieve (C-variable3 =  rel2.attr1, C-variable4 = rel2.attr2)
##        where rel2.attr3 = C-variable1
##      {
                /** C - program statements **/
##      }
## }
```

Unfortunately, nested retrieve statements are unavailable in EQUEL. If they were available, they would be used frequently when encoding data from a data base into a format. For example, consider the format specification given in Figure 9.3, and the corresponding encoder given in Figure 9.4(a) which would have been constructed using a nested retrieve statement.

Figure 9.3

```
«file»           : «object list»
«object list»    : «object» «line list»
                 | «object list» «object» «line list»
«line list»      : «line segment» | «line list» «line segment»
«object»         : OBJECT «object name» «newline»
«line segment»   : LINE «line x1» «line y1» «line x2» «line y2»
                   «newline»
```

Figure 9.3(a) : BNF specification of a format file



Relation	*Attributes*
object	objname, line name
line seg	line name, x1, y1, x2, and y2

Figure 9.3(b) : Entity - Relationship model and relations

Applying the encoder given in Figure 9.4(a) to the relations in Figure 9.4(b) would produce the file shown in Figure 9.4(c).

Figure 9.4

```
gen_file()

/**
 **     variables object_name, ln_name, ln_x1, ln_y1, ln_x2, and
 **     ln_y2 are previously defined
 **/

{
## retrieve (object_name = object.objname,
##           ln_name = object.line_name)

## {
       printf("OBJECT %s\n", object_name);

##     retrieve (ln_x1 = line_seg.x1, ln_y1 = line_seg.y1,
##                  ln_x2 = line_seg.x2, ln_y2 = line_seg.y)
##        where line_seg.line_name = ln_name

##     {
            printf("LINE %d %lf %lf\n",
                        ln_x1,ln_y1,ln_x2, ln_y2);

##     }
## }
```

Figure 9.4(a) : format encoder

object	
objname	*ln name*
ObjA	LnA
ObjA	LnB
ObjA	LnC

line				
ln name	*x1*	*y1*	*x2*	*y2*
LnA	2.2	1.3	1.1	1.4
LnB	1.1	1.4	1.7	2.1
LnC	1.7	2.1	2.2	1.3

Figure 9.4(b)

```
OBJECT ObjA
LINE 2.2 1.3 1.1 1.4
LINE 1.1 1.4 1.7 2.1
LINE 1.7 2.1 2.2 1.3
```

Figure 9.4(c)

Because the nested retrieve statement is an illegal EQUEL construct, the outer retrieve statement must be completed before the nested retrieve statement can be executed. Data retrieved by the outer statement must, therefore, be stored in a temporary data structure. Data in this structure is used by what was the second, nested, retrieve statement to fetch data from the data base.

Thus, a nested retrieve statement can be replaced with the following fragment of code:

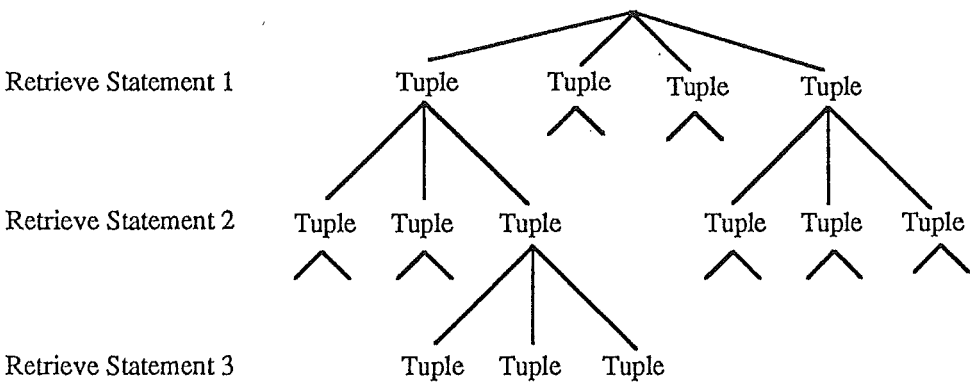
```
## retrieve (C-variable1 = rel1.attr1 , C-variable2 = rel1.attr2 )
## {
        store(C-variable1)
## }

For each stored value of C-variable1 do {

##      retrieve (C-variable3 =  rel2.attr1, C-variable4 = rel2.attr2)
##        where rel2.attr3 = C-variable1
##      {
                /** C - program statements **/
##      }
}
```

The relationship between tuples retrieved by the first statement and tuples retrieved by statements nested within this statement can be represented by a multi-way tree, accommodating an arbitrary level of nesting. A tree structure is defined as[Horowitz and Sahni 1983]:

> *a finite set of one or more nodes such that: (i) there is a specially designated node called the* **root;** *(ii) the remaining nodes are partitioned into* $n \geq 0$ *disjoint sets* $T_1....T_n$ *where each of these sets is a tree.* $T_1....T_n$ *are called* **subtrees** *of the root.*

In Figure 9.5, many tuples are fetched by the second retrieve statement for each tuple fetched by the first, and with a third retrieve statement nested within the second, many tuples may be retrieved for each tuple fetched by the second retrieve statement.

Figure 9.5



9.3. Generating Format Encoders

In Chapter 7, a description is given of how to generate format decoders according to the BNF specification of the file structure using the Yacc software tool. In this section a description is given of an *encoder generator* that processes the same, or a similar, BNF specification to generate format encoders. An implementation of this tool has been started, although limited time has prevented a working system from being completed.

The specification of the encoder to be generated is given using a BNF notation that is purposely similar to that processed by Yacc for generating format decoders. The reason for this similarity is to enable the same specification to be used by both tools, or for one specification to be easily derived from the other. To simplify implementation of the prototype encoder generator, however, specification of a list of items uses a different notation for to that employed by Yacc which is illustrated as follows. In the left hand column is a Yacc rule for specifying a list of items, and in the right hand column the equivalent notation used by the encoder generator:

```
list    :
        |       list item
        ;
```

```
list    :       { item }
        ;
```

The Yacc BNF notation is also expanded to include the following features. A metalinguistic variable of the form

relation.attribute

indicates that at that point in the file structure, a value of the specified attribute is to be substituted in much the same way as a token in a Yacc grammar indicates that a particular sequence of symbols is expected at that point in the file. For example, the encoder specification:

```
file  :     { rel1.attr1 } ;
```

indicates that the desired file consists of a list of values taken from attribute attr1 of relation rel1. The second feature introduced to the BNF specification processed by the new tool is that of directly specifying a sequence of symbols to be placed in the file. Extending the previous example so that each value of attribute occurs on individual lines, with each line being tagged with the symbols "DATA", the encoder specification would be:

```
file :      { "DATA" rel1.attr1 "\n" } ;
```

The EQUEL program generated for the above rule consists of a parametrized retrieve statement, with a sequence of calls to the standard output routine printf() to print a line of the desired form. An example is given in Figure 9.7 of a BNF description and the corresponding program generated by the encoder generator.

Figure 9.7

```
file : { "OBJECT " rel1.attr1 "," rel1.attr2 "\n" } ;
```

Figure 9.7(a) : BNF of desired file structure

```
main()
{
##   char *argv[20];
##   double dvar;
##   long ivar;
##   float fvar;
##   char svar[255];

##   ingres "kiwi::enctest"

     argv[0] = (char *)&ivar;
     argv[1] = svar;

## retrieve( param( "%i4 = rel1.attr1,%c = rel1.attr2", argv))
##{
     printf("OBJECT ");
     printf("%d", ivar);
     printf(",");
     printf("%s",svar);
     printf("\n");
##}
}
```

Figure 9.7(b) : Output of encoder generator

Rel1	
Attr1	*Attr2*
1012	Line
1013	Circle

Figure 9.7(c) : Relation to be encoded

```
OBJECT 1012,Line
OBJECT 1013,Circle
```

Figure 9.7(d) : File produced by encoder

The translation from BNF specification to an EQUEL program is performed by a translator that is generated by Yacc and Lex according to the specification files given in Appendix F. The next section discusses the structure of format encoders that would have been generated by a full working version of the encoder generator.

9.4. The Structure of Format Encoders

The general structure of a format encoder is illustrated by discussing the encoder presented in Appendix G for the character form of the BIF format. This format encoder was generated by hand to investigate the application of tree data structures for producing the desired file. The encoder that was implemented is structured as follows.

Data is encoded into the desired file structure in two stages:

1) data is stored in a tree as it is retrieved from the data base,

2) once the tree is complete, the output file is produced.

Correspondingly, the main algorithm for the format encoder is as follows:

```
/** stage (1) **/
      Init_tree( Tree );
      while ( tree_incomplete( Tree )  )
            build_tree( Tree );
/** Stage (2) **/
      produce_file( Tree );
```

The tree is constructed by the function build_tree(), which traverses the existing tree and invokes a make_subtree function pointed to by each node. A make_subtree function consists of a retrieve statement. The main body of this statement creates a node for each retrieved tuple and, as well as storing the data retrieved in the node, the function assigns the make_subtree(), and print_data() functions for that node. After invocation, the make_subtree function pointed to by the node is replaced by a function that does nothing, and once all nodes in the tree point to this empty make_subtree function, the tree is complete. The function incomplete_tree(), therefore, returns true if, on traversing the tree, it finds a non-empty make_subtree function.

In the case of the BIF format encoder, the Init_tree() function initializes the root of the tree with the following: data values for the drawing record of the file; a print_data function to produce this record; and the make_subtree function objecttree(), which retrieves the identifiers of the objects that will be in the file.

The build_tree function traverses the tree and invokes the objecttree() function which generates a subtree consisting of nodes that are individually assigned: an object identifier; a pointer to the make_subtree function called prmtree(); and a pointer to the print_data function called object(). After invocation, the objecttree() function is replaced by the empty() function so that in future traversals of the tree by the build_tree() function, the objecttree() function is not invoked.

Because of the existence of the prmtree function in the tree, the incomplete_tree function returns true and the build_tree() is invoked again. On this traversal of the tree, each occurrence of the prmtree() function results in the construction of a subtree that consists of nodes containing: the identifier and type of the primitives that define the object; and a make_subtree function and a print_data function that is appropriate for the type of the primitive.

The above sequence of function invocations continues until all the necessary data is retrieved from the data base and stored in the tree. Once the construction of the tree is complete, the produce_file() function is invoked. This function traverses the tree, invoking the print_data function pointed to by each node. The output from the print_data functions is the data from the data base structured in the manner required by the target format.

In practice, the use of a tree to implement nested retrieve statements is unnecessarily inefficient, and because the data set being processed is potentially large, it is impractical to build the entire tree in memory and then produce the file. A more space-efficient implementation strategy is needed. Fortunately, it is possible to integrate the construction of the tree and production of the file in such a way that only those branches of the tree necessary for producing the next part of the file are kept in memory. Thus a stack of retrieved data pending processing is kept, using at most the storage of one node for each level of the tree. Due to insufficient time, this strategy was not incorporated in the prototype.

Chapter Ten

Conclusions

There are many diverse and complex transfer file formats for geographic data. The translation of data from one format to another may be difficult and require considerable effort. The objectives set in this thesis were:

1) To minimise the effort involved in achieving a data translation, and

2) To maximise the quality of the translation that is achieved.

The extent to which these objectives are attainable with the following three strategies was discussed:

1) the ring interfacing strategy,

2) the interchange format interfacing strategy,

3) the individual interfacing strategy.

The ring interface was discarded because of the high potential for translating data inaccurately. The interchange format interfacing strategy has become widely established, but the value of this strategy is undermined by the definition of several different interchange formats. The individual interfacing strategy is the ideal solution, but an efficient method of implementing any interface is required for this strategy to be practical.

The author has defined a model of data translation that consists of three phases:

1) the *decode* phase, which extracts data from the source format, and places it into an equivalent relational data model,

2) the *translate* phase, which reorganises data from the source relational data model into the target relational data model,

3) the *encode* phase, which retrieves data from the relational data model and encodes it into the target format.

The modular construction of a translation process allows the development of re-useable software modules. After implementing a decoder and encoder for a format, they are used in any data translation irrespective of the other format involved. In this way, implementing the individual interfacing strategy has been reduced to implementing a translation algorithm for the translate phase of any source to target format translation pair.

The translation algorithm is specified using a relational query language provided by a relational data base management system such as INGRES, and the author's experience with INGRES indicates that specifying algorithms using QUEL can be achieved easily. Use of a system such as POSTGRES, where the standard relational data model is extended to include spatial data types and operators, will facilitate the concise definition of translation algorithms.

The effort of implementing a translation process is further reduced by generating some of the necessary software modules directly from the specification of the translation process. The syntactic structure of an implementation method for a format was specified using BNF which is processed, using software construction tools such as Yacc and Lex, to generate the format decoder and encoder. All three phases were implemented and tested for a variety of formats, although the automation of encoder generation is incomplete due to time constraints.

This thesis has shown that it is practical, and desirable, to implement the individual interfacing strategy through the use of: a three phase translation process; software tools such as Yacc and Lex; and a relational data base management system such as INGRES. In doing so, a reduction can be achieved in the effort needed to provide geographic interfaces for geographic data exchange. Further, the quality of the data translation is improved as a consequence of being able to use the individual interfacing strategy.

Acknowledgements

I wish to thank my supervisor Prof. John Penny whose interest and enthusiasm in my research, and my writing style, was greatly valued. I would also like to thank other staff of the Computer Science department at Canterbury who have offered advice during the preparation of this thesis. Bruce Mckenzie for discussion on generating parsers with Yacc and Lex, Neville Churcher for discussion on the use of relational data base management systems, and in particular Tim Bell who assisted in the final draft of this thesis.

Finally, I would like to express deep gratitude for the love and concern of my Parents who supported me while I was studying.

Bibliography

BS 6690 : 1986 / ISO 8211-1985 (1986) A data descriptive file for information interchange. British Standards Institution.

Codd E. F. (1970) A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, **13**, 377 - 387.

Codd E. F. (1972) Relational Completeness of Data Base Sublanguages. *Data Base Systems,* Courant Computer Science Symposia Series, **6**.

CSIRONET (1986) Command Driven Colourmap, User's Guide. CSIRONET Reference Manual No 19, Version 1, Graphics System Section, CSIRONET.

Data Interchange Standards. AURISA news, June 1985.

Date, C. J. (1986) An Introduction to Database Systems. **1** (Fourth Ed.)

DCDSTF (Digital Cartographic Data Standards Task Force) (1988) Spatial Data Transfer Specification, Volume 1 - The Standard. *The American Cartographer*, **15**,21-144. Volume 2 - Initial Entity and Attribute Definitions. *The American Cartographer*, **15**, 143 (microfiche).

Fosnight, E. A. and J. W. van Roessel (1985) Vector Data Interfacing at the EROS Data Center; RIM to ARC/INFO and Related Interfaces. EROS Data Center, Sioux Falls, South Dakota 57198

GDS (1984) Binary Interface File (BIF) Format. GDS Reference Manual.

GeoVision (1986) Data Translation Guide. GeoVision Reference Manual.

Goldschlager, L. and A. Lister (1988) Computer Science A Modern Introduction. Second Ed.

Guting, R. H. (1988) Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems. Computational Geomerty and its Applications, *Lecture Notes in Computer Science*, **333**.

Held G.D., M. R. Stonebraker and E. Wong (1975) INGRES- A relational data base system. AFIPS conference proceedings, **44**, 409-416

Horowitz, E. and S. Sahni (1983) Fundamentals of Data Structures

Hutchinson, M. F. (1981) MAPROJ - A computer map projection system, Division of land use research technical paper No. 39. Commonwealth scientific and industrial research organization, Australia.

INGRES (1986) INGRES Release 5.0 for the UNIX Operating System Volume 3, Relational Technology

Intergraph (1986) Standard Interchange Format (SIF) Command Language Implementation Guide (8.8.2).

International Organization for Standardization (ISO) 1986. Information processing - Text and office systems - Standard Generalized Markup Language (SGML). ISO 8879-1986 (E), International Organization for Standardization Oct 1986.

International Organization for Standardization (ISO) 1986. Information processing - Specification for a Data Descriptive File for Information Interchange. ISO 8211-1985.

Johnson, S. C. (1979) Yacc: Yet Another Compiler-Compiler. Unix Time-Sharing System: Unix Programmer's Manual, Seventh Edition, Volume 2B.

Kalash, J. L. Rodgin, Z. Fong, J. Anton (1986) INGRES - Reference manual (Version 8)

Kernighan, B. W. and D. M. Richie (1978) The C Programming Language

Lesk, M. E. and E. Schmidt (1979) Lex - A Lexical Analyzer Generator. Unix Time-Sharing System: Unix Programmer's Manual, Seventh Edition, Volume 2B.

Mamrak, S. A., M. J. Kaelbling, C. K. Nicholas, and M. Share (1987) A Software Architecture for Supporting the Exchange of Electronic Manuscripts. *Communications of the ACM*, **30**, 408 - 414.

Martin, J. and C. McClure (1986) Diagramming Techniques for Analysts and Programmers.

Naur, P. (Editor) (1963) Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, **6**, 1-17.

Nyerges, T. (Editor) (1984) Digital Cartographic Data Standards: Alternatives in Data Organization. Progress Report of the Workgroup on Data Organization of the National Committe for Digital Cartographic Data Standards, Report 4, January.

Penny J. P. (1986) Relational methods for format conversion of map data. *Cartography*, **15**, 26-34.

Penny, J. P., G. C. Ewing and R. T. Pascoe (1989) Aspects of Interactive Systems for Computer Mapping. Twelth Australian Computer Science Conference, Wollongong

Peuquet, D. (1984) A Conceptual Framework and Comparison of Spatial Data Models. *Cartographica*, **21**, 66-113

Rich, C. and R. C. Waters (1988) Automatic Programming: Myths and Prospects. *Computer* IEEE.

Samet (1984) The Quadtree and related Hierarchical Data Structures. A*CM Computing Surveys*, **16**, 187-260

Shu N. C., B. C. Housel, R. E. Taylor, S. P. Ghosh, and V. Y. Lum (1977) EXPRESS: A Data EXtraction, Processing, and REStructuring System. *ACM Trans. on Database Systems*, **2**, 134-174.

Smith, H. C. (1985) Database design: composing fully normalized tables from a rigorous dependency diagram.*Communications of the ACM*, **28**, 826.

Steel T. B. Jr. (1960) UNCOL: The myth and the fact. *Annual Review in Automatic Programming*, **2**, 325-344.

Stevens, R.J., A.F.Lehar, and F. H. Preston (1983) Manipulation and presentation of multidimensional image data using the Peano scan. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, **5**, 520-526.

Stonebraker, M. and L. A. Rowe (1986) The Design of POSTGRES. Proceedings ACM SIGMOD 1986, International Conference on Management of Data, Washington D.C.

Stonebraker, M., C. Williams (1975) An approach to implementing a geodata system, *proceedings of the workshop on databases for interactive design*, 67-77

Stonebraker, M., E. Wong, P. Kreps, and G. Held (1976) The design and implementation of INGRES. *ACM Transactions on Database Systems*, **1**, 189-222.

Tanenbaum, A. S. (1978) Implications of Structured Programming for Machine Architecture, *Communications of the ACM*, **21**, 237 - 246.

Tanenbaum, A. S., H. van Straveren, E. G. Keizer, and J. W. Stevenson (1983) A Practical Tool Kit for Making Portable Compilers. *Communications of the ACM*, **26**, 654 - 660.

Taylor, R. W. 1982 Experiences Using Generalized Data Translation Techniques for Database Interchange. *Computers & Standards* **1**, 111-118

van Berkel, P. (1987) Proposal for a Standard for the Interchange of Digital Land Data. A Discussion Document, Land Information New Zealand (LINZ) Support Group.

van Roessel, J. W. (1987) Design of a spatial data structure using the relational normal forms. *Int. J. Geographical Information Systems,* **1**,33-50.

van Roessel, J. W. and Fosnight, E. A. (1985) A relational Approach to vector data structure conversion. *Proc. Auto-carto 7 Conf.* Washington

van Roessel, J., D. Bankers, V. Connochioli, S. Doescher, G. Fosnight, M. Wehde, and D.Taylor (1986) Vector Data Structure Conversion at the EROS Data Center. Final Report (Draft), Phase I, EROS Data Center.

Appendix A

Format Implementation Specifications

This appendix contains an implementation specification for the GeoVision GINA format, the character form of the BIF format, and the Colourmap format. Throughout this Appendix the metalinguistic variable «empty» indicates the null string, that is:

«empty» ::=

A.1 The GeoVision GINA Format

A.1.1 BNF Specification

«GINA file» ::= «physical format file» «database header file» «primary schema file» «feature file» «polygon relationship file» «indirect attributes file» «data base trailer file»

«physical format file» ::= UDB-START «block format» «block length» «max block size»

«block format» := BLOCKED | UNBLOCKED

«block length» := F | VARIABLE

«max block size» ::= UNSIGNED_INT

«database header file» ::= «dbf header record» «description record» «coordinate system record» «extent record» «layer record list» «network record list»

«dbf header record» ::= UDB-HEADER «GINA version number» «database name» «database description»

«GINA version number» ::= REAL

«database name» ::= CHAR

«database description»::= STRING

«description record» ::= «empty» | DESCR DATE TIME «GINA user name» «comment»

«GINA user name» ::= CHARACTER

«comment» ::= «empty» | STRING

Note :: The question marks in the following indicate that the structure was not clearly defined at this point of the format description.

«coordinate system record» ::= «empty»
| COORD-SYS «mercator crds» «coordinate units» «spheroid definition» «scale factor at central meridian» «zone width» «central meridian» «false easting» «false northing»
| COORD-SYS GEOG DEGREES
| COORD-SYS GEOG DEGREES OVERRIDE «semi_major axis»«semi_minor axis»
| COORD-SYS RECT «coordinate units»
| COORD-SYS MERC «coordinate units» ??
| COORD-SYS LAMBERT «coordinate units» ??
| COORD-SYS NC_STATE «coordinate units» ??

«mercator crds» ::= TM | UTM

«coordinate units» ::= «empty» | FEET | METRES | MILLIMETRES | INCHES | KILOMETRES

«spheroid definition» ::= CLARK | IAU | OVERRIDE «semi_major axis» «semi_minor axis»

«semi_major axis» ::= REAL

«semi_minor axis» ::= REAL

«scale factor at central meridian» ::= REAL

«zone width» ::= UNSIGNED_INT

«central meridian» ::= INTEGER

«false easting» ::= «empty» | UNSIGNED_INT

«false northing» ::= «empty» | UNSIGNED_INT

«extent record» ::= «empty»
| EXTENT «min x or long coordinate» «min y or lat coordinate» «max x or long coordinate» «max y or lat coordinate»

«min x or long coordinate» ::= REAL | INTEGER

«min y or lat coordinate» ::= REAL | INTEGER

«max x or long coordinate» ::= REAL | INTEGER

«max y or lat coordinate» ::= REAL | INTEGER

«layer record list» ::= «empty» | «layer record list» «layer record»

«layer record» ::= LAYER «layer number» «layer name» «layer description»

«layer number» ::= UNSIGNED_INT

«layer name» ::= CHARACTER

«layer description» ::= STRING

«network record list» ::= «empty | «network record list» «network record»

«network record» ::= NETWORK «network number» «network name» «network type» «layer number» «network description»

«network number» ::= UNSIGNED_INT

«network name» ::= CHAR

«network type» ::= PLGN | L

«network description» ::= STRING

«primary schema file» ::= UDB-PRIMARY «table definition list»

«table definition list» ::= «table definition» | «table definition list» «table definition»

«table definition» ::= «fixed table record» «fixed field definition list» «feature code list»
| «free table record» «free field definition list» «feature code list»

«fixed table record» ::= TABLE «table name» FIXED

«free table record» ::= TABLE «table name» FREE

«table name» ::= CHAR

«fixed field definition list» ::= «fixed field record» | «fixed field definition list» «fixed field record»

«fixed field record» ::= FIELD «field name» «field description» «index flag» «unique flag» «null flag» «description of attribute» «start column of field» «number of columns field occupies»

«free field definition list» ::= «free field record» | «free field definition list» «free field record»

«free field record» ::= FIELD «field name» «field description» «index flag» «unique flag» «null flag» «description of attribute»

«field name» ::= CHAR

«field description» ::= CHR «number of characters» | NUM «number of digits» COMMA «number of decimal digits»

«number of characters» ::= UNSIGNED_INT

«number of digits» ::= UNSIGNED_INT

«number of decimal digits» ::= UNSIGNED_INT

«index flag» ::= «empty» | I

«unique flag» ::= «empty» | UNQ

«null flag» ::= «empty» | N

«description of attribute» ::= STRING

«start column of field» ::= UNSIGNED_INT

«number of columns field occupies» ::= UNSIGNED_INT

«feature code list» ::= «feature code record» | «feature code list» «feature code record»

«feature code record» ::= FEAT_CODE «low feature code» «high feature code»

«low feature code» ::= «feature code»

«high feature code» ::= «feature code»

«feature file» ::= UDB-FEAT «feature definition list»

«feature definition list» ::= «feature record» «coordinate list» «label definition» «attributes»

«feature record» ::= FEAT «feature sequence number» «feature code» «layer number» «network number» «type of feature» «coordinate definition» «graphic parameter 1» «graphic parameter 2» «flow direction»

«feature sequence number» ::= UNSIGNED_INT

«feature code» ::= CHAR

«type of feature» ::= PLGN | N | L

«coordinate definition» ::= XY | XYZ | XYCZ «z»

«graphic parameter 1» ::= CHAR

«graphic parameter 2» ::= CHAR

«flow direction» ::= INTEGER

«coordinate list» ::= «coordinate record» | «coordinate list» «coordinate record»

«coordinate record» ::= COOR «coordinate pair list»

«coordinate pair list» ::= «2d coordinate list» | «3d coordinate list»

«2d coordinate list» ::= «x» COMMA «y» | «2d coordinate list» «x» COMMA «y»

«3d coordinate list» ::= «x» COMMA «y» COMMA «z»
| «3d coordinate list» «x» COMMA «y» COMMA «z»

«x» ::= REAL | INTEGER

«y» ::= REAL | INTEGER

«z» ::= REAL | INTEGER

«label definition» ::= «empty» | «label record» | «text record» | «label record» «text record»

«label record» ::= LABEL «label number» «label x position» «label y position» «orientation flag» «orientation angle»

«label number» ::= UNSIGNED_INT

«label x position» ::= REAL | INTEGER

«label y position» ::= REAL | INTEGER

«orientation flag» ::= «empty» | HORZ | REL

«orientation angle» ::= «empty» | REAL

«text record» ::= TEXT «text character string»

«text character string» ::= STRING

«attributes» ::= «empty» | «attribute record» «attribute continuation list»

«attribute record» ::= ATTR «attribute value list»

«attribute continuation list» ::= «empty» | «attribute continuation list» «attribute continuation record»

«attribute continuation record» ::= ATCR «attribute value list»

«attribute value list» ::= «attribute value» | «attribute value list» «attribute value»

«attribute value» ::= *as defined by primary schema file*

«polygon relationships file» ::= UDB-PLGN «relationship record list»

«relationship record list» ::= «relationship record» | «relationship record list» «relationship record»

«relationship record» ::= POLY «line sequence number» «left / right flag» «interior / exterior flag»

«line sequence number» ::= UNSIGNED_INT

«left / right flag» ::= L | R | QU

«interior / exterior flag» ::= I | E

«indirect attributes file» ::= UDB-INDRT «table definition list» «attributes»

«database trailer file» ::= UDB-END

Lexical Symbol	*Regular Expression*
ATCR	"atcr"
ATTR	"attr"
BLOCKED	"b"
CHAR	[A-Za-z~!@#$%..]+
CHR	"char"
CLARK	"Clark-1866"
COMMA	","
COOR	"coor"
COORD-SYS	[Cc][Oo][Oo][Rr][Dd]-[Ss][Yy][Ss]
DATE	[0-3][0-9][01][0-9] [0-9][0-9][0-9][0-9]
DEGREES	"degrees"
DESCR	"descr"
E	"e"
EXTENT	"extent"
F	"f"
FEAT	"feat"
FEAT_CODE	"fc"
FEET	"feet"
FIELD	"field"
FIXED	"fixed"
FREE	"free"
GEOG	"geographicals"
HORZ	"horizontal"
I	"i"
I	"i"
IAU	"IAU-1965"
INCHES	"inches"
INTEGER	[+-]?[0-9]+
KILOMETRES	"kilometres"
L	"l"
LABEL	"label"
LAMBERT	"Lambert"
LAYER	"layer"
MERC	"Mercator"
METRES	"metres"
MILLIMETRES	"millimetres"
N	"n"
NC_STATE	"NC state plane"
NETWORK	"network"
NUM	"num"
OVERRIDE	"override"

Lexical Symbol	*Regular Expression*
PLGN	"p"
POLY	"poly"
QU	"?"
R	"r"
RECT	"rectangular"
REAL	([+-]?[0-9]+"."[0-9]* ([E][+-][0-9][0-9])*) | ([+-]?[0-9]*"."[0-9]+ ([E][+-][0-9][0-9])*)
REL	"relative"
STRING	\"([A-Za-z~!@#$..]? [ ])+\"
TABLE	"table"
TEXT	"text"
TIME	[0-2][0-9]":"[0-5][0-9] ":"[0-5][0-9]
TM	"tm"
UDB-END	[Uu][Dd][Bb]-[Ee][Nn][Dd]
UDB-HEADER	[Uu][Dd][Bb]-[Hh][Ee][Aa][Dd] [Ee][Rr]
UDB-INDRT	[Uu][Dd][Bb]-[Ii][Nn][Dd][Ii][Rr] [Ee][Cc][Tt]
UDB-PLGN	[Uu][Dd][Bb][-[Pp][Oo][Ll][Yy] [Gg][Oo][Nn]
UDB-PRIMARY	[Uu][Dd][Bb]-[Pp][Rr][Ii][Mm] [Aa][Rr][Yy]
UDB-START	[Uu][Dd][Bb]-[Ss][Tt][Aa][Rr][Tt]
UDB_FEAT	[Uu][Dd][Bb]-[Ff][Ee][Aa][Tt] [Uu][Rr][Ee]
UNBLOCKED	"u"
UNQ	"u"
UNSIGNED_INT	[0-9]+
UTM	"utm"
VARIABLE	"v"
XY	"xy"
XYCZ	"xycz"
XYZ	"xyz"

A.2 The BIF Format

A.2.1 BNF Specification

«datafile» ::= «empty» | «drawing» «object list»

«empty» ::= *(that is, the null string of symbols)*

«drawing» ::=DRAWING REAL REAL REAL REAL

«object list» ::= «empty» | «object list» «object»

«object» := «name» «hook» «primitive list»

«name» ::= NAME «ocd value»

«ocd value» ::= «compound value» | «ocd value» COLON «compound value»

«compound value» ::= «a value» | «compound value» «a value»

«a value» ::= INTEGER | REAL | STRING

«hook» ::= HOOK REAL REAL

«primitive list» ::= «primitive» | «primitive list» «primitive»

«primitive» ::= «line» «segment list» | «circle» | «text» «text line list» | «points» «point list»

«line» ::= LINE REAL REAL «compound value»

«segment list» ::= «to» | «arc» | «segment list» «to» | «segment list» «arc»

«to» ::= TO REAL REAL

«arc» ::= ARC REAL REAL REAL

«circle» ::= CIRCLE REAL REAL REAL STRING

«text» ::= TEXT REAL REAL REAL REAL REAL REAL STRING «a value»

«text line list» ::= «chars» | «text line list» «chars»

«chars» ::= CHARS STRING

«points» ::= POINT STRING

«point list» ::= «at» | «point list» «at»

«at» ::= AT REAL REAL

A.2.2 Lexicon

Lexical Symbol	*Regular Expression*
ARC	"arc"
AT	"at"
CHARS	"chars"
CIRCLE	"circle"
COLON	":"
DRAWING	"drawing"
HOOK	"hook"
INTEGER	[+-]?[0-9]+
LINE	"Line"
NAME	"Name"

Lexical Symbol	*Regular Expression*
POINT	"point"
REAL	([+-]?[0-9]+"."[0-9]* ([E][+-][0-9][0-9])*) | ([+-]?[0-9]*"."[0-9]+ ([E][+-][0-9][0-9])*)
STRING	([\\\-A-Za-z'$#!%^& ~?/]+[0-9]*)+
TEXT	"text"
TO	"to"

A.3 The Colourmap Format

The following specification defines three types of Colourmap data files: zone files, line overlay files, and standard attribute files.

A.3.1 BNF Specification

«datafile» ::= «comment» NEWLINE «header» «data»

«data» ::= «name list» «attribute sectn» | «zone part» «poly part» «seg part» | «seg part»

«zone part» ::= ZONES NEWLINE «num zones» NEWLINE «zone defn list»

«poly part» ::= «empty» | POLYGONS NEWLINE «num polygons» NEWLINE «polygon defn list»

«seg part» ::= «empty» | SEGMENTS NEWLINE «num segments» NEWLINE «segment defn list»

«zone defn list» ::= «z defn» | «zone defn list» «z defn»

«polygon defn list» ::= «poly defn» | «polygon defn list» «poly defn»

«segment defn list» ::= «segment defn» | «segment defn list» «segment defn»

«z defn» ::= «zone id» «poly in zone» NEWLINE «poly id list» NEWLINE

«poly defn» ::= «poly defn hdr» «x poly label» «y poly label» «polygon area» NEWLINE «seg id list» NEWLINE
| «poly defn hdr» «x poly label» «y poly label» NEWLINE «seg id list» NEWLINE
| «poly defn hdr» «polygon area» NEWLINE «seg id list» NEWLINE

«poly defn hdr» ::= «poly id» «zone id» «segs in poly» «dsp level» NEWLINE «x min poly regn» «x max poly regn» «y min poly regn» «y max poly regn»

«segment defn» ::= «segment id» «seg left zone» «seg right zone» «points in segment» NEWLINE «xy defn list» NEWLINE

«seg id list» ::= «empty» | «seg id list» NEWLINE | «seg id list» INTEGER

«poly id list» ::= «empty» | «poly id list» NEWLINE | «poly id list» INTEGER

«xy defn list» ::= REAL REAL REAL REAL | «xy defn list» NEWLINE | «xy defn list» REAL REAL

«header» ::= MAP NEWLINE «map window» «map projection»
| «num attr» «num names» «missing dat val» NEWLINE

«map window» ::= «X min regn» «X max regn» «Y min regn» «Y max regn» NEWLINE

«map projection» ::= «map proj code» «zone type» NEWLINE | «map proj code» NEWLINE

«name list» ::= «empty» | «name list» «name» NEWLINE | «name list» «name»

«attribute sectn» ::= «attribute list»

«attribute list» ::= «empty» | «attribute list» «attribute»

«attribute» ::= «attr defn» «attr val list»

«attr defn» ::= «description» «units» NEWLINE

«attr val list» ::= «empty» | «attr val list» «attr value» NEWLINE | «attr val list» «attr value»

«attr value» ::= REAL

«comment» ::= «string»

«polygon area» ::= REAL

«description» ::= «string»

«dsp level» ::= INTEGER

«map proj code»::= INTEGER

«poly in zone» ::= INTEGER

«segs in poly» ::= INTEGER

«points in segment» ::= INTEGER

«missing dat val» ::= REAL

«name» ::= «string»

«num attr» ::= INTEGER

«num segments» ::= INTEGER

«num names» ::= INTEGER

«num polygons» ::= INTEGER

«num zones» ::= INTEGER

«poly id» ::= INTEGER

«segment id» ::= INTEGER

«seg left zone» ::= «string»

«seg right zone» ::= «string»

«units» ::= «string»

«X min regn» ::= REAL

«X max regn» ::= REAL

«x max poly regn» ::=REAL

«x min poly regn» ::= REAL

«x poly label» ::= REAL

«y max poly regn» ::= REAL

«Y max regn» ::= REAL

«Y min regn» ::= REAL

«y min poly regn» ::= REAL

«y poly label» ::= REAL

«zone id» ::= «string»

«zone type» ::= INTEGER

«string» ::= STRING

A.3.2 Lexicon

Lexical Symbol	*Regular Expression*
INTEGER	[\t]+[+-]?[0-9]+[\t]*
MAP	"MAP"[\t]*\n
NEWLINE	"\n"
POLYGONS	"POLYGONS"["\n"
REAL	[\t]*[+-]?[0-9]+"." [0-9]*([E][-+][0-9] [0-9])?[\t]* | [\t]*[+-]?[0-9]*"." [0-9]+([E][-+][0-9] 0-9])?[\t]*
SEGMENTS	"SEGMENTS"[\t]*
STRING	(([\\-A-Za-z0-9'$#!%^&~`:?/] (".")?)+[\t]*)+
ZONES	"ZONES"[\t]*\n

Appendix B

Format Decoders

B.1 The Colormap Format Decoder

B.1.1 Yacc Definition file

```
%{
    /** $Header$ **/

#include <stdio.h>

#include "eql.h"

#define maxrgflg 5
#define INITRG(A) rgcnt[++rgflg] = A;
#define DECRG rgcnt[rgflg]--;

#define newtkn(A) yylval.t.code = A;
#define itv(A) A.v.ival
#define dtv(A) A.v.dval
#define stv(A) A.v.sval

int rgcnt[maxrgflg], rgflg = 0;

char *sf;               /** source datafile name            **/

int datasize = 80;      /** used by lex to determine size of string
**/
                        /** (initialised for size of comment)
**/

extern int eqlappend(), eqlreplace();
extern char *strsave();

double atof();

char *zid;
int pid, sid, nattr, nname, attrseq;
%}

%start datafile

%union {
    struct TKN {
      int code;
      union val {
      int ival;
      double dval;
      char *sval;
      }v;
    }t;
```

```
    int filetype, itmseq;
    struct PD {
      int pid, sip;
      double xminregn, xmaxregn, yminregn, ymaxregn;
    } polydefn;
  }

%type <filetype> data
%type <polydefn> poly.defn.hdr
%type <t> _zone.id_ _poly.in.zone_ _poly.id_ _segs.in.poly_
_num.attr_
%type <t> _dsp.level_ _x.min.poly.regn_ _x.max.poly.regn_
%type <t> _y.min.poly.regn_ _y.max.poly.regn_ _x.poly.label_
%type <t> _y.poly.label_      _polygon.area__segment.id_
%type <t>  _seg.left.zone_ _seg.right.zone_    _points.in.segment_
%type <t> _num.zones_ _num.polygons_ _num.segments_  string
%type <t> _num.names_ _missing.dat.val_ _name_ _description_
%type <t> _units__attr.value_ _comment_

%type <itmseq> seg.id.list xy.defn.list name.list attribute.list
%type <itmseq> attr.val.list

%token <t> MAP POLYGONS ZONES SEGMENTS LINES POINTS NEWLINE
%token <t> INTEGER REAL STRING

/** system tokens **/

%token <t> EORG SETSIZE STRCONT

%%
/*-------------------------------------------------------------------*/
/*                Notation                                           */
/*                ========                                           */
/*                                                                   */
/*    _symbol.name_     => Lexical symbol                            */
/*    TOKEN       => a reserved symbol/data type                     */
/*    symbol.name       => yacc non-terminal symbol                  */
/*                                                                   */
/*-------------------------------------------------------------------*/

datafile :  _comment_ NEWLINE header data
      {
          eqlreplace("cm_dss","srcfile",sf,"tffkey",$4,EOAL, EOQL);
          free(stv($1));
      }
      ;

data :      names attribute.sectn
      {
          $$ = 10;
      }
      |     zone.part poly.part seg.part
      {
          $$ = 0;           /** Zone file **/
      }
      |     line.part seg.part
      {
          $$ = 2;           /** line file **/
      }
      |     seg.part
      {
```

```
            $$ = 4;             /** line overlay **/
        }


        |     point.part
        {
            $$ = 1; /** Either A: site, B: name or, C: marker
overlay**/
        };

zone.part :       ZONES NEWLINE
              _num.zones_ NEWLINE
                { INITRG(itv($3)); } zone.defn.list EORG
          {
            eqlreplace("cm_dss", "nzones", itv($3), EOAL, EOQL);
        }
              ;

poly.part :
        |     POLYGONS NEWLINE
              _num.polygons_ NEWLINE
              { INITRG(itv($3)); } polygon.defn.list EORG
          {
            eqlreplace("cm_dss", "npolygons", itv($3), EOAL, EOQL);
        }
        ;

seg.part :
        |     SEGMENTS NEWLINE
              _num.segments_ NEWLINE
              { INITRG(itv($3)); } segment.defn.list EORG
         {
            eqlreplace("cm_dss", "nareachains", itv($3), EOAL, EOQL);

        }
        ;

line.part   : LINES
              ;

point.part  : POINTS
              ;

map.window :      REAL REAL REAL REAL NEWLINE
        {
            eqlreplace("cm_dss", "xminregn", dtv($1), "xmaxregn",
dtv($2),
              "yminregn", dtv($3), "ymaxregn", dtv($4), EOAL, EOQL);
        }
        ;

map.projection :
              INTEGER INTEGER NEWLINE
        {
           eqlreplace("cm_dss","projection",itv($1),"zonetype",itv($2),
                    EOAL, EOQL);
        }
        |     INTEGER NEWLINE
        {
            eqlreplace("cm_dss", "projection", itv($1), EOAL, EOQL);
        }
        ;
```

```
zone.defn.list :
            z.defn
      |     zone.defn.list z.defn
      ;

polygon.defn.list :
            poly.defn
      |     polygon.defn.list poly.defn
      ;

segment.defn.list :
            segment.defn
      |     segment.defn.list segment.defn
      ;

z.defn :    _zone.id_ _poly.in.zone_ NEWLINE
          { INITRG(itv($2)); zid = strsave(stv($1));}
            poly.id.list EORG NEWLINE
      {
          eqlappend("cm_zone", "zoneid", zid, "cmptpol", itv($2),
                EOAL);
          DECRG; free(stv($1)); free(zid);
      }
      ;

poly.defn :       poly.defn.hdr
            _x.poly.label_ _y.poly.label_ _polygon.area_ NEWLINE
          { INITRG($1.sip); pid  = $1.pid; }
            seg.id.list EORG NEWLINE
      {
          char *s1;

          s1 = (char *) malloc(200); *s1 = '\0';
          sprintf(s1, " cm_polygon.polid = %d", $1.pid);

          eqlreplace("cm_polygon", "xlabel", dtv($2), "ylabel",
dtv($3),
                   "area", dtv($4), EOAL, s1, EOQL);
          DECRG; free(s1);
      }
      |     poly.defn.hdr _x.poly.label_ _y.poly.label_ NEWLINE
          { INITRG($1.sip); } seg.id.list EORG NEWLINE
      {
          /**
           ** calculate area
           **/

          double area;
          char *s1;

          area = ($1.xmaxregn - $1.xminregn) *
            ($1.ymaxregn - $1.yminregn);
          s1 = (char *) malloc(200); *s1 = '\0';
          sprintf(s1, " cm_polygon.polid = %d", $1.pid);
          eqlreplace("cm_polygon","xlabel",dtv($2),"ylabel",dtv($3),
                   "area", area, EOAL, s1, EOQL);
          DECRG; free(s1);
      }


      |       poly.defn.hdr _polygon.area_ NEWLINE
```

```
        { INITRG($1.sip); } seg.id.list EORG NEWLINE
    {
        /**
         ** calculate centroid
         **/

        double cx, cy;
        char *s1;

        cx = ($1.xminregn + $1.xmaxregn) / 2;
        cy = ($1.yminregn + $1.ymaxregn) / 2;

        s1 = (char *) malloc(200); *s1 = '\0';
        sprintf(s1, " cm_polygon.polid = %d", $1.pid);

        eqlreplace("cm_polygon", "xlabel", cx, "ylabel", cy,
                "area", dtv($2), EOAL, s1, EOQL);
        DECRG; free(s1);
    }
    ;

poly.defn.hdr :  _poly.id_ _zone.id_ _segs.in.poly_ _dsp.level_
NEWLINE
          _x.min.poly.regn_ _x.max.poly.regn_
          _y.min.poly.regn_ _y.max.poly.regn_
    {
        eqlappend("cm_polygon", "polid", itv($1), "enclzoneid",
stv($2),
                "cmptachain", itv($3),"dsplevel", itv($4),
                "xminregn", dtv($6), "xmaxregn", dtv($7),
                "yminregn", dtv($8), "ymaxregn", dtv($9),
                EOAL);

        $$.pid = itv($1); $$.sip = itv($3);
        $$.xminregn = dtv($6); $$.xmaxregn = dtv($7);
        $$.yminregn = dtv($8); $$.ymaxregn = dtv($9);
        free(stv($2));
    }
    ;

segment.defn :   _segment.id_ _seg.left.zone_ _seg.right.zone_
          _points.in.segment_ NEWLINE
        { INITRG(itv($4)); sid = itv($1); }
          xy.defn.list EORG NEWLINE
    {
        int i;

        eqlappend("cm_areachain", "achainid", itv($1),
"leftzoneid", stv($2),
                "rightzoneid", stv($3), "cmptpt", itv($4), EOAL);
        DECRG; free(stv($2)); free(stv($3));
    }
        ;
```

```
seg.id.list :
        {
            $$ = 0;
        }
        |     seg.id.list NEWLINE
        {
            $$ = $1;
        }

        |     seg.id.list INTEGER
        {
            eqlappend("cm_polachain","polid",pid,"seq", $1, "achainid",
                    itv($2), EOAL);

            $$ = $1 + 1; DECRG;
        }
        ;

poly.id.list :
        |     poly.id.list NEWLINE

          |   poly.id.list INTEGER
        {
            eqlappend("cm_zonepol","zoneid",zid,"polid",itv($2), EOAL);
            DECRG;
        }
        ;

xy.defn.list :    REAL REAL REAL REAL
        {
            eqlappend("cm_achainpt","achainid",sid,"seq",0,"x",dtv($1),
                    "y", dtv($2), EOAL);
            eqlappend("cm_achainpt","achainid",sid,"seq",1,"x",dtv($3),
                    "y", dtv($4), EOAL);

            $$ = 2; DECRG; DECRG;
        }
        |     xy.defn.list NEWLINE
        {
            $$ = $1;
        }
        |     xy.defn.list REAL REAL
        {
           eqlappend("cm_achainpt","achainid",sid,"seq",$1,"x",dtv($2),
                    "y", dtv($3), EOAL);

            $$ = $1 + 1; DECRG;
        }
        ;

header      :     MAP NEWLINE map.window map.projection
        |     _num.attr_ _num.names_ _missing.dat.val_ NEWLINE
        {
            nattr = itv($1); nname = itv($2);

            eqlappend("cm_attrdss","nattributes",nattr,"nnames", nname,
                    "missdatval", dtv($3), EOAL);
        }
        ;
```

```
names       :     { INITRG(nname); } name.list EORG
      ;
name.list :
      {
          $$ = 0;
      }
      |     name.list _name_ NEWLINE
      {
          eqlappend("cm_objname", "nameseq",$1,"name",stv($2), EOAL);
          $$ = $1 + 1; DECRG; free(stv($2));
      }
      |     name.list _name_
      {
          eqlappend("cm_objname","nameseq",$1,"name",stv($2), EOAL);
          $$ = $1 + 1; DECRG; free(stv($2));
      }
      ;

attribute.sectn :
            { INITRG(nattr); } attribute.list EORG
      ;

attribute.list :
      {
          $$ = 0;
      }
      |     attribute.list { attrseq = $1; datasize = 30; } attribute
      {
          $$ = $1 + 1; DECRG;
      }
      ;

attribute :       attr.defn { INITRG(nname); } attr.val.list EORG
      ;

attr.defn :       _description_ _units_ NEWLINE
      {
          eqlappend("cm_attrdefn","atid",attrseq,"description",
                  stv($1), "units", stv($3), EOAL);
          free(stv($1)); free(stv($3));
      }
      ;

attr.val.list :
      {
           $$ = 0;
      }
      |     attr.val.list _attr.value_ NEWLINE
      {
          eqlappend("cm_attrvalue", "atid", attrseq, "nameseq", $1,
                  "value", dtv($2), EOAL);
          $$ = $1 + 1; DECRG;
      }
      |     attr.val.list _attr.value_
      {
          eqlappend("cm_attrvalue", "atid", attrseq, "nameseq", $1,
                  "value", dtv($2), EOAL);
          $$ = $1 + 1; DECRG;
      }
      ;
```

```
/*-----------------------------------------------*/
/*          Lexicon definitions                  */
/*-----------------------------------------------*/

_attr.value_ :    REAL
      { $$ = $1; }
      ;
_comment_ :       { datasize = 80; } string
      {
            eqlappend("cm_dss", "description", stv($2), EOAL);
      }
      ;

_polygon.area_ :
            REAL
      { $$ = $1; }
      ;

_description_ :
            { datasize = 30; } string
      { $$ = $2; }
      ;

_dsp.level_ :     INTEGER
      { $$ = $1; }
      ;

_poly.in.zone_ :
            INTEGER
      { $$ = $1; }
      ;

_segs.in.poly_ :
            INTEGER
      { $$ = $1; }
      ;

_points.in.segment_ :
            INTEGER
      { $$ = $1; }
      ;

_missing.dat.val_ :
            REAL
      { $$ = $1; }
      ;

_name_ :    { datasize = 10; } string
      { $$ = $2; }
      ;

_num.attr_ :      INTEGER
      { $$ = $1; }
      ;

_num.segments_    :
            INTEGER
      { $$ = $1; }
      ;
```

```
_num.names_          :
            INTEGER
      { $$ = $1; }
      ;

_num.polygons_ :
            INTEGER
      { $$ = $1; }
      ;

_num.zones_ :     INTEGER
      { $$ = $1; }
      ;

_poly.id_ :       INTEGER
      { $$ = $1; }
      ;

_segment.id_ :    INTEGER
        { $$ = $1; }
      ;

_seg.left.zone_   :
          { datasize = 10; } string
      { $$ = $2; }
      ;

_seg.right.zone_ :
          { datasize = 10; } string
      { $$ = $2; }
      ;

_units_ :   { datasize = 10; } string
      { $$ = $2; }
      ;

_x.max.poly.regn_ :
            REAL
      { $$ = $1; }
      ;

_x.min.poly.regn_ :
            REAL
      { $$ = $1; }
      ;

_x.poly.label_ :
            REAL
      { $$ = $1; }
      ;

_y.max.poly.regn_ :
            REAL
      { $$ = $1; }
      ;

_y.min.poly.regn_ :
            REAL
      { $$ = $1; }
      ;
```

```
_y.poly.label_ :
            REAL
      { $$ = $1; }
      ;

_zone.id_ :  { datasize = 10;} string
      { $$ = $2; }
      ;

/*-------------------------------------------------*/
/*          General Constructs                     */
/*-------------------------------------------------*/

string      :     SETSIZE STRING
      {
          $$ = $2;
      }
      ;

%%
#include "lex.c"

yyerror(s)
     char *s;
{
  printf("yytext:%s:\n", yytext);
}
main()

{
    FILE *freopen();

    sf = (char *) malloc(30);

    printf("Source data filename ?              "); fflush(stdout);
    scanf("%s", sf); getc(stdin);

    freopen(sf, "r", stdin);

    eqlopen("kiwi::cm", "-c10");
    yyparse();
    eqlclose();
}
```

B.1.2 Lex definition File

```
%a 5000
%o 9000
%{
#define input() ((!(rgcnt[rgflg]) && rgflg) ? (rgflg--, '@') :\
(((yytchar=yysptr>yysbuf?U(*--yysptr):\
getc(yyin))==10?(yylineno++,yytchar):yytchar)==EOF?0:yytchar))

#define REAL_SIZE 10
#define INT_SIZE  10
int sizeset = 0;
%}
D           [0-9]
E           [E][-+]{D}{D}
W           [\t ]
```

```
BYTE        [\,\-A-Za-z0-9'$#!%^&~`:?/]

%%
{W}*\n                  { return( NEWLINE ); }

MAP{W}*\n               { yyless(yyleng - 1); return( MAP ); }

POLYGONS{W}*\n          { yyless(yyleng - 1); return( POLYGONS ); }

ZONES{W}*\n             { yyless(yyleng - 1); return( ZONES );}

SEGMENTS{W}*\n          { yyless(yyleng - 1); return( SEGMENTS );}

LINES{W}*\n             { yyless(yyleng -1); return( LINES );}

POINTS{W}*\n            { yyless(yyleng - 1); return( POINTS );}

{W}*[+-]?{D}+"."{D}*({E})?{W}*      |
{W}*[+-]?{D}*"."{D}+({E})?{W}*  {
                        if (yyleng > REAL_SIZE)
                              yyless(yyleng - (yyleng - REAL_SIZE));
                        newtkn( REAL );
                        dtv(yylval.t) = atof(yytext);
                        return( REAL );
                    }
{W}+[+-]?{D}+{W}*       {
                        if (yyleng > INT_SIZE)
                              yyless(yyleng - (yyleng - INT_SIZE));
                        newtkn( INTEGER );
                        itv(yylval.t) = atoi(yytext);
                        return( INTEGER );
                    }
(({BYTE}(".")?)+{W}*)+  {
                        if (!sizeset) {
                              yyless(0); sizeset++;
                              return( SETSIZE );
                        }
                        else if (yyleng > datasize)
                              yyless(datasize);
                        newtkn( STRING );
                        stv(yylval.t) = strsave(yytext);
                        sizeset = 0;
                        return( STRING );
                    }
@                   { return( EORG ); }
%%
```

B.2 The GINA Format Decoder

The definition files given here only defines the feat, and coor record structures. This was all that was necessary to achieve the desired translation.

B.2.1 Yacc Definition file

```
%{
    /** $Header$ **/

#include <stdio.h>

#include "eql.h"

#define maxrgflg 5
#define INITRG(A) rgcnt[++rgflg] = A;
#define DECRG rgcnt[rgflg]--;

#define newtkn(A) yylval.t.code = A;
#define itv(A) A.v.ival
#define dtv(A) A.v.dval
#define stv(A) A.v.sval
#define qtv(A) A.v.qval

int rgcnt[maxrgflg], rgflg = 0;

char *sf;    /** source datafile name  **/

    int fid, crdseq;

    extern int eqlappend();
    extern char *strsave();

    double atof();
%}

%start datafile

%union {
    struct TKN {
      int code;
      union val {
      int ival;
      double dval;
      char *sval;
      }v;
    }t;
    int itmseq;
  }

%type <t> feat.id feat.code layer.nmbr ntwrk.nmbr feat.type crd.type
%type <t> grph.param1 grph.param2 flw.drtn x.crd y.crd
/** lex tokens **/

%token <t> REAL INTEGER STRING NEWLINE
%token <t> UDBFEAT FEAT COOR
```

```
/** system tokens **/

%token <t> EORG SETSIZE STRCONT

%%
/*-------------------------------------------------------------------*/
/*                          Notation                                 */
/*                          ========                                 */
/*                                                                   */
/*          _symbol.name_=> Lexical symbol                           */
/*          TOKEN         => a reserved symbol/data type             */
/*          symbol.name       => yacc non-terminal symbol            */
/*                                                                   */
/*-------------------------------------------------------------------*/

datafile    :     UDBFEAT NEWLINE
                  feature.lst
            ;
feature.lst       :     feature
            |     feature.lst feature
            ;
feature     :     feature.dfn
                  crd.dfn
            ;
feature.dfn       :     FEAT feat.id feat.code layer.nmbr ntwrk.nmbr
feat.type
                  crd.type grph.param1 grph.param2 flw.drtn NEWLINE
            {
                eqlappend("dosfeat", "fid", itv($2), "fc", stv($3),
                  "layer", itv($4), "network", itv($5), "feattype",
stv($6),
                  "crdtype", stv($7), "gp1", dtv($8), "gp2", dtv($9),
                  "flw", itv($10), EOAL);

                fid = itv($2); crdseq = 0;
            }
            ;
crd.dfn     :     COOR crd.lst NEWLINE
            |     crd.dfn COOR crd.lst NEWLINE
            ;
crd.lst     :     crds
            |     crd.lst crds
            ;
crds        :     x.crd y.crd
            {
                  eqlappend("doscrds", "fid", fid, "seq", crdseq,
                  "x", dtv($1), "y", dtv($2), "z",0.0, EOAL);

                  crdseq++;
            }
            ;
/*-----------------------------------------------*/
/*          Lexical equivalence                  */
/*-----------------------------------------------*/

feat.id     :     INTEGER
            ;
feat.code   :     STRING
            ;
layer.nmbr  :     INTEGER
            ;
```

```
ntwrk.nmbr  :     INTEGER
            ;
feat.type   :     STRING
            ;
crd.type    :     STRING
            ;
grph.param1       :     REAL
            ;
grph.param2       :     REAL
            ;
flw.drtn    :     INTEGER
            ;
x.crd       :     REAL
            |     INTEGER
            {
                  double x;
                  x = itv($1);
                  dtv($1) = x;
            }
            ;
y.crd       :     REAL
            |     INTEGER
            {
                  double y;
                  y = itv($1);
                  dtv($1) = y;
            }
            ;
/*---------------------------------------------*/
/*          General Constructs                 */
/*---------------------------------------------*/

%%
#include "lex.c"

yyerror(s)
     char *s;
{
  printf("yytext:%s:\n", yytext);
}
main()

{
    eqlopen("kiwi::cm", "-c10");
    yyparse();
    eqlclose();
}
```

B.2.2 Lex Definition File

```
%o4000
%{

#define input() ((!(rgcnt[rgflg]) && rgflg) ? (rgflg--, '@') :
(((yytchar=yysptr>yysbuf?U(*--
yysptr):getc(yyin))==10?(yylineno++,yytchar):yytchar)==EOF?0:yytchar)
)

%}
```

```
D               [0-9]
E               [E][-+](D)(D)
W               [\t ]
CHAR            [\,\-A-Za-z'$#!%^&~`:?/_]

%%
(W)*\n              { return( NEWLINE ); }

udb-feature         { return( UDBFEAT ); }

feat                { return( FEAT ); }
coor                { return( COOR ); }

(W)*[+-]?(D)+"."(D)*((E))?(W)*      |
(W)*[+-]?(D)*"."(D)+((E))?(W)*  {
                            newtkn( REAL );
                            dtv(yylval.t) = atof(yytext);
                            return( REAL );
                    }

(W)*[+-]?(D)+(W)*           {
                            newtkn( INTEGER );
                            itv(yylval.t) = atoi(yytext);
                            return( INTEGER );
                    }

\"((CHAR)(D)*(W)*)*\" {
                            newtkn( STRING );
                            stv(yylval.t) = strsave(yytext+1);
                            stv(yylval.t)[yyleng-1] = '\0';
                            return( STRING );
                    }
((CHAR)(D)*)+       {
                            newtkn( STRING );
                            stv(yylval.t) = strsave(yytext);
                            return( STRING );
                    }

@                   { return( EORG ); }
%%
/*-------------------------------------------------------------*/
```

B.3 The BIF Decoder

B.3.1 Yacc Definition file

```
%{
#include <stdio.h>
#include "que.h"
#include "yacc.defn.h"
#include "eql.h"

    int cmpobjid=0, objectid= 0, txtlnseq=0, objseq=0, xyseq=0;

    extern char *strsave();

    double atof();
    char *pntstyle, *malloc();

%}
%start datafile

%union {
    QOBJPTR t;
    int itmseq, ival;
    double dval;
    char *sval, *ocd[4];
}
%type <ival> integer.value
%type <dval> real.value
%type <sval> string.value a.value compound.value
%type <ocd> ocd.value

%token <t> EORG NEWLINE DRAWING NAME HOOK LINE TO ARC CIRCLE TEXT
%token <t> CHARS POINT AT COLON

/** NB: the following must agree with that in que.h **/

%token <t> STRING 1001 INTEGER 1003 REAL 1002

%%
datafile :
      |     datafile drawing NEWLINE
      |     datafile name NEWLINE
      |     datafile hook NEWLINE
      |     datafile line NEWLINE
      |     datafile to NEWLINE
      |     datafile arc NEWLINE
      |     datafile circle NEWLINE
      |     datafile text NEWLINE
      |     datafile chars NEWLINE
      |     datafile points NEWLINE
      |     datafile at NEWLINE
      ;
drawing :   DRAWING real.value real.value real.value real.value
      {
            eqlappend("drw", "dunit", $2, "dcenx", $3, "dceny", $4,
                  "dsize", $5, EOAL);
      }
      ;
```

```
name :            NAME ocd.value
      {
            eqlappend("cmpobjnm", "cmpobjid", ++cmpobjid, "ocd",
$2[0],
                  "dpso", $2[1], "othr1", $2[2], "othr2", $2[3],
EOAL);
            objseq=0;
      }
      ;
ocd.value :       compound.value
      {
            $$[0] = $1;
            $$[1] = strsave("-");
            $$[2] = strsave("-");
            $$[3] = strsave("-");
      }
      |     compound.value COLON compound.value
      {
            $$[0] = $1;
            $$[1] = $3;
            $$[2] = strsave("-");
            $$[3] = strsave("-");
      }
      |     compound.value COLON compound.value COLON compound.value
      {
            $$[0] = $1;
            $$[1] = $3;
            $$[2] = $5;
            $$[3] = strsave("-");
      }
      |     compound.value COLON compound.value COLON
            compound.value COLON compound.value
      {
            $$[0] = $1;
            $$[1] = $3;
            $$[2] = $5;
            $$[3] = $7;
      }
      ;
compound.value :
            a.value
      {
            $$ = $1;
      }
      |     compound.value a.value
      {
            $$ = (char *)malloc(strlen($1) + strlen($2) + 1);
            sprintf($$, "%s%s", $1,$2);
      }
      ;
hook :            HOOK real.value real.value
      {
            eqlreplace("cmpobjnm", "hookx", $2, "hooky", $3, EOAL,
                  "cmpobjid", cmpobjid, EOQL);
      }
      ;
```

```
line :          LINE real.value real.value compound.value
      {
            xyseq = 0;
            eqlappend("cmpobjdfn", "cmpobjid", cmpobjid, "objectid",
                  ++objectid, "objecttype", 1, "seq", objseq++,
                  "style", $4, EOAL);
            eqlappend("curvedfn","curveid", objectid, "seq", xyseq++,
                  "x", $2, "y", $3, "bulge", 0.0, EOAL);
      }
      ;
to :        TO real.value real.value
      {
            eqlappend("curvedfn", "curveid", objectid,"seq", xyseq++,
                  "x", $2, "y", $3, "bulge", 0.0, EOAL);
      }
      ;
arc :       ARC real.value real.value real.value
      {
            eqlappend("curvedfn", "curveid", objectid,"seq", xyseq++,
                  "x", $2, "y", $3, "bulge", $4, EOAL);
      }
      ;
circle :    CIRCLE real.value real.value real.value string.value
      {
            eqlappend("cmpobjdfn", "cmpobjid", cmpobjid, "objectid",
                  ++objectid, "objecttype", 2, "seq", objseq++,
                  "style", $5, EOAL);
            eqlappend("circledfn", "circleid", objectid, "cntrx", $2,
                  "cntry", $3, "radius", $4, EOAL);
      }
      ;
text :          TEXT real.value real.value real.value real.value
real.value
            real.value string.value a.value
      {
            eqlappend("cmpobjdfn", "cmpobjid", cmpobjid, "objectid",
                  ++objectid, "objecttype", 3, "seq", objseq++,
                  "style", $9, EOAL);
            eqlappend("txtblk", "txtblkid", objectid, "orgnx", $2,
                  "orgny", $3, "width", $4, "height", $5,
                  "sinrot", $6, "cosrot", $7, "just", $8, EOAL);
            txtlnseq=0;
      }
      ;
a.value :   integer.value
      {
            $$ = (char *) malloc(20);
            sprintf($$, "%d", $1);
      }
      |     real.value
      {
            $$ = (char *) malloc(20);
            sprintf($$, "%lf", $1);
      }
      |     string.value
      {
            $$ = $1;
      }
      ;
```

```
chars :           CHARS
      {
            eqlappend("txtln", "txtblkid", objectid, "seq",
                  txtlnseq++, "line", stv($1), EOAL);
      }
      ;
points :    POINT string.value
      {
            pntstyle = $2;
      }
      ;
at :        AT real.value real.value
      {
            eqlappend("cmpobjdfn", "cmpobjid", cmpobjid, "objectid",
                  ++objectid, "objecttype", 0, "seq", objseq++,
                  "style", pntstyle, EOAL);
            eqlappend("pntdfn", "pntid", objectid, "x", $2,
                  "y", $3, EOAL);
      }
      ;
integer.value : INTEGER
        { $$ = itv($1); }
        ;

real.value :    REAL
        { $$ = dtv($1); }
        ;
string.value :    STRING
      { $$ = strsave(stv($1)); }
      ;
%%
#include "lex.c"

yyerror(s)
    char *s;
{
    printf("ERROR with yytext = :%s:\n", yytext);
}
main(argc, argv)
      int argc;
      char *argv[];
{
    char *sf;
    FILE *freopen();

      if ((argc < 3) || (argc > 3)) {
            fprintf(stderr, "Usage: impbif server::database
filename\n");
            exit();
      }

    freopen(argv[2], "r", stdin);

    eqlopen(argv[1], "-c10");
    eqlappend("filename", "file", argv[2], EOAL);
    yyparse();
    eqlclose();
}
```

```
%a 5000
%o 9000
%{

#define input() ((!(rgcnt[rgflg]) && rgflg) ? (rgflg--, '@') :
(((yytchar=yysptr>yysbuf?U(*--
yysptr):getc(yyin))==10?(yylineno++,yytchar):yytchar)==EOF?0:yytchar)
)
#define REAL_SIZE 10
#define INT_SIZE  10

int sizeset = 0;
%}

D           [0-9]
E           [E][-+]{D}{D}
W           [\t ]
ALPHA       [\.\,\-A-Za-z'$#!%^&~`?/]

%%
{W}*\n              { return( NEWLINE ); }
":"                 { return( COLON); }
drawing             {return( DRAWING );}
name                { return( NAME ); }
hook                { return( HOOK ); }
line                { return( LINE ); }
to                  { return( TO ); }
arc                 { return( ARC ); }
circle              { return( CIRCLE ); }
text                { return( TEXT ); }
points              { return( POINT );}
at                  { return( AT ); }
"chars "(({ALPHA})*[ 0-9]*)+ {
            newtkn( STRING );
            stv(yylval.t) = strsave(yytext+6);
            return( CHARS );
      }
{W}*[+-]?{D}+"."{D}*({E})?{W}*      |
{W}*[+-]?{D}*"."{D}+({E})?{W}*  {
                        newtkn( REAL );
                        dtv(yylval.t) = atof(yytext);
                        return( REAL );
                    }
{W}*[+-]?{D}+       {
                        newtkn( INTEGER );
                        itv(yylval.t) = atoi(yytext);
                        return( INTEGER );
                    }
(({ALPHA})+[0-9]*)+     {
    newtkn( STRING );
    stv(yylval.t) = strsave(yytext);
    return( STRING );
}
@       { return( EORG ); }
%%
/*-------------------------------------------------------------*/
```

Appendix C

A Simplified INGRES Interface

This appendix contains a listing of the functions used by a source decoder to append and replace data in an INGRES data base. These functions, provide a measure of independence to a source decoder from changes to the relational data base and generally enhance the interface between a decoder and it's data base.

Also included with the listing of the eqlappend(), and eqlreplace() functions, is a listing of other functions that are used to implement these two functions, and a listing of a set of functions for operating a multi-level hash table. Preceeding these listings is a listing of two include files that are used by many of the above functions.

C.1. eql.h

```
/**
 **   The header file for the EQL functions
 **
 **/
#define MAXATTR 30
#define EOAL "@EQLEND"
#define EOQL "@EQLQLEND"

struct ATTRDEFN {
    char *relid, *attrid, typ;
    int sz;
};
```

C.2. hash.h

```
#define HASHSIZE 47
#define Hhashtab 0
#define Hattrdefn 1

typedef struct tblentry {
    char *name;
    union defpart {
      struct ATTRDEFN *ad;
      struct tblentry **ht;
    } def;
    struct tblentry *next;
} *TBL_ENTRY, **HASH_TABLE;
```

C.3. eqlappend()

```
#include "eql.h"
#include <varargs.h>
#include <stdio.h>

extern struct ATTRDEFN *getattype();
extern char *atctrl(), *strsave();

eqlappend(va_alist)

    va_dcl

      /** This function appends attributes to a relation
       **
       ** argument structure :
       **
       **     (relation, { attr_name, attr_value }, EOAL)
       **/

{
##  char *relid, *tl, *av[ MAXATTR ];

    va_list ap;

    char *s, *ac, *attr_name;
    int nav;
    struct ATTRDEFN *ad;

    char *sval;
    int ival;
    double dval;

    nav = 0;
    va_start(ap);

    relid = va_arg(ap, char *);

    attr_name = va_arg(ap, char *);

    while (strcmp(attr_name, EOAL)) {

      ad = getattype(relid, attr_name);
      ac = atctrl(ad);
      if (nav) {
          s = (char *) malloc(2+ strlen(tl) + strlen(ac));
          sprintf(s, "%s,%s", tl, ac);
          free(tl); free(ac); tl = s;
      }
      else
          tl = ac; ac = NULL;
```

```
        switch(ad->typ) {
            case 'c' : {
              av[nav++] = va_arg(ap, char *);
              break;
            }
            case 'i' : {
              av[nav++] = (char *) &va_arg(ap, int);
              break;
            }
            case 'f' : {
              av[nav++] = (char *) &va_arg(ap, double);
              break;
            }
            other :;
        }
    attr_name = va_arg(ap, char *);
    }

##  append relid(param(tl, av))
free(tl);
}
```

C.4. eqlreplace()

```
#include "eql.h"
#include <varargs.h>

extern struct ATTRDEFN *getattype();
extern char *atctrl(), *strsave();

eqlreplace(va_alist)

    va_dcl

      /**
       ** This function replaces attribute values in an ingres
       ** relation according to the given qualification
       **
       ** the argument structure is :
       **
       **    (relid, { attr_name, attr_value }+,EOAL, [ qual, ] EOQL)
       **
       **  where {..}+ indicates contents repeated 1 or more times
       **  and [ ] indicates contents optional.
       **
       **  EOAL, and EOQL are defined in eql.h
       **/

{
##  char *relid, *tl, *av[ MAXATTR ], *qual;

    va_list ap;

    char *s, *ac, *attr_name;
    int nav;
    struct ATTRDEFN *ad;

    char *sval;
    int ival;
    double dval;
```

```
    nav = 0;
    va_start(ap);

    relid = va_arg(ap, char *);
    attr_name = va_arg(ap, char *);

    while (strcmp(attr_name, EOAL)) {

      ad = getattype(relid, attr_name);
      ac = atctrl(ad);
      if (nav) {
          s = (char *) malloc(2+ strlen(tl) + strlen(ac));
          sprintf(s, "%s,%s", tl, ac);
          free(tl); free(ac); tl = s;
      }
      else
          tl = ac;

      switch(ad->typ) {
          case 'c' : {
            av[nav++] = va_arg(ap, char *);
            break;
          }
          case 'i' : {
            av[nav++] = (char *) &va_arg(ap, int);
            break;
          }
          case 'f' : {
            av[nav++] = (char *) &va_arg(ap, double);
            break;
          }
          other :;
      }
    attr_name = va_arg(ap, char *);
    }

      qual = va_arg(ap, char *);

    if (strcmp(qual, EOQL))
##      replace relid(param(tl, av)) where qual
    else
##      replace relid(param(tl, av))
      free(tl);
}
```

C.5. atctrl()

```
#include "eql.h"

extern struct ATTRDEFN *getattype();

char  *atctrl(ad)

    struct ATTRDEFN *ad;
/**
 ** creates a string to be used in a target list for the attribute
 ** defined by ad
 **/

#define STROH 8  /** NULL(1) + ' = %'(4) + type(1) + size(2) **/
{
    char *s, *t;

    s = (char *) malloc( STROH + strlen(ad->attrid) );


    if (ad->typ != 'c')
      sprintf(s, "%s = %%%c%d", ad->attrid, ad->typ, ad->sz);
    else
      sprintf(s, "%s = %%%c", ad->attrid, ad->typ);

    return(s);
}
```

C.6. getattype()

```
#include <stdio.h>
#include "eql.h"
#include "hash.h"

extern char *strsave();     /** libutl/strfns.c **/

struct ATTRDEFN *getattype(rid, aid)

##    char *rid,         /** the relation name **/
##       *aid;         /** the attribute name **/



    /** ASSUMPTION: the database is open **/

    /** USAGE: returns a structure ATTRDEFN that contains the type
and **/
    /** size of the named attribute
     **
     **   opendb();
     **
     **   attrdefn = getattype(relation, attribute);
     **/
```

```
{

##  int sz, ty;

    static HASH_TABLE hashtab;
    HASH_TABLE newhash();
    struct ATTRDEFN *ad, *hlookup(), *hinstall();

    if (hashtab == NULL) hashtab = newhash();
    if ((ad = hlookup(hashtab, rid, aid)) != NULL)
      return(ad);

    ad = (struct ATTRDEFN *) malloc (sizeof(*ad));
    ad->attrid = strsave(aid);
    ad->relid = strsave(rid);

##  retrieve(ty = attribute.attfrmt, sz = attribute.attfrml)
##  where attribute.attrelid = rid and attribute.attname = aid

    ad->sz = sz;

      switch(ty) {
            case 30 :
                  ad->typ = 'i';
                  break;

            case 31 :
                  ad->typ = 'f';
                  break;

            case 32 :
                  ad->typ = 'c';
                  break;

            default : ;
      }

  hinstall(hashtab, ad); /** ad = NULL **/

#ifdef GDBG
    printf(">>getattype: rid:%s: aid:%s: ty:%c: sz:%d:\n", ad->relid,
          ad->attrid, ad->typ, ad->sz);
#endif

  return(ad);
}
```

C.7. hash functions

The following hash functions implement a hash table. Each entry in this table describes an attribute that has been used in an eqlreplace or an eqlappend. The performance of these functions is improved because, if the attribute is used again, it is unnecessary to retrieve this description a second time from the data base. To find an entry in the hash table, the relation name is used to find the attributes stored for this relation, then the attribute name is hashed to find the description of the desired attribute.

```
#include <varargs.h>
#include <stdio.h>
#include "eql.h"
#include "hash.h"

extern char *strsave();

/**
 **====================================================================
 **/

HASH_TABLE  newhash()
{
    char * calloc();

    return( (HASH_TABLE) calloc( HASHSIZE, sizeof(TBL_ENTRY)));
}
/**
 **====================================================================
 **/

newentry(typ,  ht,  ad)

    int typ;
    HASH_TABLE ht;
    struct ATTRDEFN *ad;

{
  TBL_ENTRY  tp, duptp, find();
  HASH_TABLE newhash();
  int hv, hash();

  tp = (struct tblentry *) malloc( sizeof( *tp));

  switch(typ) {

      case Hhashtab : {

        tp->name = ad->relid;
        tp->def.ht = newhash();
        newentry(Hattrdefn, tp->def.ht, ad);
        break;
      }
      case Hattrdefn : {

        if ((duptp = find(ad->attrid, ht)) == NULL) {
            tp->name = ad->attrid ;
            tp->def.ad = ad; ad = NULL;
        }
        else {
            free(tp);
            duptp->def.ad->sz = ad->sz; duptp->def.ad->typ = ad->typ;
            return(0);
        }
        break;
      }
  default :;
  }
  hv = hash(tp->name);
  tp->next = ht[hv]; ht[hv]=tp;
  return(0);
}
```

```
/**
 **====================================================================
 **/

hash(s)

    char *s;

    /**
     ** usage : hash(keyvalue);
     **/

{
    int hashval;

    for (hashval = 0; *s != '\0';)
      hashval += *s++;
    return(hashval % HASHSIZE);
}

/**
 **====================================================================
 **/

TBL_ENTRY  find(s,  ht)

    char *s;
    HASH_TABLE ht;

     /**
      **  Locate an entry in one level of the hash data structure
      **/

{
  TBL_ENTRY tp;

  if (ht == NULL) return(NULL);

  for (tp = ht[hash(s)]; tp != NULL; tp = tp->next)

    if (strcmp(s, tp->name) == 0) /** found entry **/
      return(tp);

  return(NULL);
}
/**
 **====================================================================
 **/

struct  ATTRDEFN  *hlookup(ht,  r,  a)

    /** look for s in hashtab **/

    HASH_TABLE ht;
    char *r, *a;

    /**
     ** usage: hash_tab_entry = lookup(keyvalue);
     **
     ** purpose: Retrieve a pointer to the ATTRDEFN record created
     **          for attribute a, in relation r
     **/
```

```
{
    TBL_ENTRY te1, te2, find();

    if ((te1 = find(r, ht)) != NULL)

      if ((te2 = find(a, te1->def.ht))!= NULL)
      return(te2->def.ad);

    return(NULL);        /** not found **/
}

/**
 **==========================================================
 **/

struct  ATTRDEFN  *hinstall(ht,  ad)

    /** put name,def into hash table **/

    HASH_TABLE ht;
    struct ATTRDEFN *ad;

    /**
     ** usage:
     **    entry_hash_tab = install(hashtable,keyvalue definition);
     **
     ** purpose: Create a new entry for the attribute defined by ad
     **          in the hash data structure
     **/

{
    TBL_ENTRY te1, find();
    int hash();

    if ((te1 = find(ad->relid, ht)) == NULL)

      /** new relation entry **/

      newentry(Hhashtab, ht, ad);

    else    /** new attribute entry **/

      newentry(Hattrdefn, te1->def.ht, ad);
#ifdef HDEBUG
    printf(". . . installed\n");
#endif
}

/**
 **==========================================================
 **/
clearhash(ht)
    HASH_TABLE ht;
{
    int i;

    for (i =0; i < HASHSIZE; i++)
      if (ht[i] != NULL)
          clearentry(Hhashtab, ht[i]);
    free(ht);
}
```

```
clearentry(typ,  entry)
    int typ;
    TBL_ENTRY entry;

{
    int j;

    if (entry != NULL) {
/**
 ** note that entry -> name is a pointer to either ad->relid or
 ** ad->attrid. When either of the last two are freed the first is
 **  implicitly freed.
 **/
      switch(typ) {
          case Hattrdefn : {
            free(entry->def.ad->relid);
            free(entry->def.ad->attrid);
            free(entry->def.ad);
            break;
          }
          case Hhashtab : {
            for (j=0; j<HASHSIZE; j++)
                if (entry->def.ht[j] != NULL)
                  clearentry(Hattrdefn, entry->def.ht[j]);
            free(entry->def.ht);
            break;
          }
          default :
            break;
      }
      clearentry(typ, entry->next);
      free(entry);
    }
}
```

Appendix D

Lexical Analysis of Repeating Groups

This appendix contains the variable declarations and data structures used within the lexical analyser for processing enumerated repeating groups in a format. It is applicable for lexical analysers that are generated using the Lex software tool [Lesk and Schmidt 1979].

The following variables are declared:

```
#define maxrgflg 5
#define INITRG(A) rgcnt[++rgflg] = A;
#define DECRG rgcnt[rgflg]--;
int rgcnt[maxrgflg], rgflg = 0;
```

Table D.1 indicates how the analyser can assertain the state of the repeating group mechanism using the values of rgflg, and rgcnt[rgflg].

Table D.1

Condition	*Value of rgflg*	*Value of rgcnt[rgflg]*
Mechanism inactive	0	0
Within incomplete repeating group	non-zero	remaining number of data objects to be parsed
Repeating group completed	non-zero	0

The lexical analyser employes the following input macro:

```
#define input() ((!(rgcnt[rgflg]) && rgflg) ? (rgflg--, '@') :
                    (((yytchar=yysptr>yysbuf ? U(*--yysptr): \
getc(yyin)) == 10 ? (yylineno++,yytchar): \
yytchar)==EOF ? 0 : yytchar))
```

When the input macro establishes that a repeating group has been completed an '@' character is inserted into the character stream. A token definition associating the '@' character to the EORG token is included within the set of token definitions supplied to Lex. Thus, by inserting the '@' into the character stream the desired EORG token will be generated by the lexical analyser. To maintain data integrity, the '@' character is absorbed out of the input character stream by the lexical analyser.

Appendix E

Example Translation Algorithm

To illustrate the use of QUEL, a query language provided by INGRES, a translation algorithm written in this language is given in § E.3. The algorithm translates data from the relational model given in § E.1 for the BIF format, into a model given in § E.2 which is used for research in the Department of Computer Science, University of Canterbury.

E.1 BIF Format Relational Data Model

Relations for storing data that is either coming from or going to the BIF format is created by the following QUEL statements.

```
create drw(dunit=f8, dcenx=f8, dceny=f8, dsize=f8)
create cmpobjnm(cmpobjid=i4, hookx = f8, hooky = f8, ocd=c10,
      dpso=c15,othr1=c15, othr2=c15)
create hookobj(objectid=i4, hookid=i4)
create cmpobjdfn(cmpobjid=i4, objectid=i4, objecttype=i4,
      seq=i4, style=c20)
create curvedfn(curveid=i4, seq=i4, x=f8, y=f8, bulge=f8)
create circledfn(circleid=i4, cntrx=f8, cntry=f8, radius=f8)
create txtblk(txtblkid=i4, orgnx=f8, orgny=f8, width=f8, height=f8,
      sinrot=f8, cosrot=f8, just=c3)
create txtln(txtblkid=i4, seq=i4, line=c250)
create pntdfn(pntid=i4, x=f8, y=f8)
create filename(file = c100)
\p\g
```

E.2 Research Relational Data Model

Relations for storing data that is being used for research is created using the following QUEL statements.

```
create layer(layname=c63, layid=i4)
create reps(layid=i4, repid=i4, minlevel=f4, maxlevel=f4)
```

```
create object(repid=i4, objid=i4, objtype=i4, minx=f4, maxx=f4,
miny=f4,maxy=f4)
create point(pointid=i4, x=f4, y=f4)
create curve(curveid=i4, seq=i4, x=f4, y=f4, bulge=f4)
create circle(circleid=i4, centx=f4, centy=f4, rad=f4)
create textblock(textid=i4, rectx=f4, recty=f4, rectw=f4, recth=f4,
      theta=f4, just=c2)
create textline(textid=i4, seq=i4, chars=c255)
\p\g
```

E.3 Translation Algorithm

Data in the relations for the BIF format (§ E.1) is translated into the relations being used for research (§ E.2) by the following translation algorithm.

```
/**
 **   ensure objectid to be unique (all objectid > 0)
 **/
retrieve tmp(maxobjid = max(object.objid)) \p\g
replace cmpobjdfn(objectid = cmpobjdfn.objectid + tmp.maxobjid) \p\g
replace curvedfn(curveid = curvedfn.curveid + tmp.maxobjid) \p\g
replace circledfn( circleid = circledfn.circleid + tmp.maxobjid) \p\g
replace pntdfn( pntid = pntdfn.pntid + tmp.maxobjid) \p\g
replace txtblk( txtblkid = txtblk.txtblkid + tmp.maxobjid) \p\g
replace txtln( txtblkid = txtln.txtblkid + tmp.maxobjid) \p\g
destroy tmp

/**
 **   (temporarily) ensure cmpobjid to be unique
 **/
retrieve tmp(maxcmpobjid = max(cmpobjlay.cmpobjid))\p\g
replace cmpobjdfn(cmpobjid = cmpobjdfn.cmpobjid + tmp.maxcmpobjid)\p\g
replace cmpobjnm(cmpobjid = cmpobjnm.cmpobjid + tmp.maxcmpobjid) \p\g
destroy tmp

append objsrc(source = "CHCHDB", file = filename.file,
      mncmpobjid = min(cmpobjdfn.cmpobjid),
      mxcmpobjid = max(cmpobjdfn.cmpobjid))
```

```
/**
 **   append cmpobj
 **/

append to cmpobjlay(cmpobjnm.all) \p\g
append to cmpobj(cmpobjid=cmpobjdfn.cmpobjid,
objid=cmpobjdfn.objectid,
      objtype=cmpobjdfn.objecttype, seq=cmpobjdfn.seq, style =
      cmpobjdfn.style) \p\g

/**   calculate minimum bounding rectangle of objects            **/

range of cdfn is curvedfn
retrieve tmp(objectid = cdfn.curveid, minx=min(cdfn.x by
cdfn.curveid),
      maxx = max(cdfn.x by cdfn.curveid), miny = min(cdfn.y by
cdfn.curveid),
      maxy = max(cdfn.y by cdfn.curveid)) \p\g
range of cdfn is circledfn
append tmp(objectid = cdfn.circleid, minx = cdfn.cntrx - cdfn.radius,
      maxx = cdfn.cntrx + cdfn.radius, miny= cdfn.cntry - cdfn.radius,
      maxy = cdfn.cntry + cdfn.radius) \p\g
append tmp(objectid = pntdfn.pntid, minx = pntdfn.x, maxx = pntdfn.x,
      miny = pntdfn.y, maxy = pntdfn.y) \p\g
range of tb is txtblk
append tmp(objectid = tb.txtblkid,
      minx = tb.orgnx - tb.height * tb.sinrot,
      maxx =tb.orgnx + tb.width * tb.cosrot, miny = tb.orgny,
      maxy = tb.orgny + tb.width* tb.sinrot + tb.height * tb.cosrot)
  where tb.cosrot >= 0 and tb.sinrot >= 0 \p\g
append tmp(objectid = tb.txtblkid,
      minx = tb.orgnx + tb.width * tb.cosrot - tb.height * tb.sinrot,
      maxx = tb.orgnx, miny = tb.orgny + tb.height * tb.cosrot,
      maxy = tb.orgny + tb.width * tb.sinrot)
  where tb.sinrot >=0 and tb.cosrot < 0 \p\g
```

```
append tmp(objectid = tb.txtblkid,
      minx = tb.orgnx + tb.width * tb.cosrot,
      maxx = tb.orgnx - tb.height*tb.sinrot,
      miny = tb.orgny + tb.width*tb.sinrot+tb.height*tb.cosrot,
      maxy = tb.orgny)
  where tb.sinrot < 0 and tb.cosrot < 0 \p\g
append tmp( objectid = tb.txtblkid, minx = tb.orgnx,
      maxx = tb.orgnx + tb.width * tb.cosrot,
      miny = tb.orgny + tb.width * tb.sinrot,
      maxy = tb.orgny + tb.height * tb.cosrot)
  where tb.sinrot < 0 and tb.cosrot >=0 \p\g

/**
 **   append to object relation
 **/

append object(objid = cmpobjdfn.objectid,
                  objtype = cmpobjdfn.objecttype)
replace object(minx = tmp.minx, maxx=tmp.maxx, miny = tmp.miny, maxy =
tmp.maxy)
  where object.objid = tmp.objectid \p\g

destroy tmp

/**
 **   append curve definitions to mapbase
 **/

append to curve(curvedfn.all)

/**
 **   append circle definitions to mapbase
 **/

append to circle(circleid = circledfn.circleid,
centx = circledfn.cntrx,
      centy = circledfn.cntry, rad = circledfn.radius)
```

```
/**
 **   append point definitions to mapbase
 **/
append to point(pointid = pntdfn.pntid, x=pntdfn.x, y = pntdfn.y)

/**
 **   append text blocks to mapbase
 **/

append to textblock(textid = txtblk.txtblkid, rectx=txtblk.orgnx,
      recty=txtblk.orgny, rectw=txtblk.width, recth=txtblk.height,
      just=txtblk.just, theta = 0.0)
/**   calculate theta, and replace in mapbase **/
/**   pi = 3.141592     **/
replace textblock(theta=3.141592 + atan(txtblk.sinrot/txtblk.cosrot))
  where textblock.textid = txtblk.txtblkid and txtblk.cosrot < 0.0
replace textblock(theta=atan(txtblk.sinrot/txtblk.cosrot))
  where textblock.textid = txtblk.txtblkid
    and txtblk.cosrot >= 0.0 and txtblk.sinrot >= 0.0
replace textblock(theta=2*3.141592+atan(txtblk.sinrot/txtblk.cosrot))
  where textblock.textid = txtblk.txtblkid
    and txtblk.sinrot < 0.0 and txtblk.cosrot >= 0.0

append to textline(textid = txtln.txtblkid, seq = txtln.seq,
      chars = txtln.line)
\p\g
```

Appendix F

Format Encoder Generator

The Yacc and Lex files given in this Appendix are used to generate the prototype generator of fomat encoders.

F.1 Yacc Definition file

```
%{
#include <stdio.h>
#include "que.h"
#include "yacc.defn.h"
#include "eql.h"

    int attr_seq = 0;

    extern char *strsave(), *retatctrl();
    extern struct ATTRDEFN *getattype();

    FILE *fp;

    double atof();
    char *dbname, *malloc();
    struct PARM_LIST *mkstrprm(), *mkattr();

%}
%start datafile

%union {
    QOBJPTR t;
    int ival;
    double dval;
    char *sval;
    struct ATTRDEFN *aval;
    struct PARM_LIST {
      char *var_list, *target_list, *prf;
      } *pval;
}

%type <sval> string.value  rule rhs
%type <pval> param_list
%type <aval> attribute

%token <t> EORG IDENT COLON L_BRCK R_BRCK DOT SEMI

/** NB: the following must agree with that in que.h **/

%token <t> STRING 1001 INTEGER 1003 REAL 1002

%%
datafile :  rule
      ;
```

```
rule  :     IDENT COLON rhs SEMI
      {
            fprintf(fp,
            "main()\n\n{\n##\tchar *argv[20];\n##\tdouble dvar;\n");
            fprintf(fp,
      "##\tlong ivar;\n##\tchar svar[255];\n##\tfloat fvar;\n\n");
            fprintf(fp, "##\tingres \"%s\"\n", dbname);
            fprintf(fp, "%s\n}\n", $3);
      }
      ;

rhs   :     L_BRCK param_list R_BRCK
      {
            $$ = (char *) malloc(45 + strlen($2->target_list) +
                  strlen($2->prf) + strlen($2->var_list)  + 1);
            sprintf($$, "%s## retrieve( param( \"%s\",
argv))\n##{\n%s\n##}\n",
                  $2->var_list, $2->target_list, $2->prf);
            attr_seq = 0;
      }
      ;

param_list :      attribute
      {
            $$ = mkattr($1, attr_seq++);
            printf("var list:>%s\n---\n", $$->var_list);
            printf("target_list:>%s\n------\n", $$->target_list);
            printf("prf:>%s\n-----\n", $$->prf);
      }
            |     string.value
      {
            $$ = mkstrprm($1);
            printf("var list:>%s\n---\n", $$->var_list);
            printf("target_list:>%s\n------\n", $$->target_list);
            printf("prf:>%s\n-----\n", $$->prf);
      }
            |     param_list attribute
      {
            struct PARM_LIST *pl;

            pl = mkattr($2, attr_seq++);
            $$ = (struct PARM_LIST *)
                  malloc( sizeof(struct PARM_LIST));
            if (strlen($1->target_list)) {
                  $$->target_list =
                        (char *) malloc(strlen($1->target_list)
                                    + strlen(pl->target_list) + 2);
                  sprintf($$->target_list, "%s,%s", $1->target_list,
                        pl->target_list);
                  free($1->target_list);
                  free(pl->target_list);
            }
            else
                  $$->target_list = pl->target_list;

            $$->var_list = (char *) malloc(strlen($1->var_list) +
                        strlen(pl->var_list) + 1);
            sprintf($$->var_list, "%s%s", $1->var_list, pl->var_list);
            free($1->var_list); free(pl->var_list);
```

```
            $$->prf = (char *)
                  malloc(strlen($1->prf)+strlen(pl->prf) +25);
            sprintf($$->prf, "%s%s", $1->prf, pl->prf);

            free($1->prf); free($1);
            free(pl->prf); free(pl);
            printf("var list:>%s\n---\n", $$->var_list);
            printf("target_list:>%s\n------\n", $$->target_list);
            printf("prf:>%s\n-----\n", $$->prf);
      }
            |     param_list string.value
      {
            struct PARM_LIST *pl;

            pl = mkstrprm($2);
            $$ = (struct PARM_LIST *)
                        malloc( sizeof(struct PARM_LIST));
            $$->target_list = $1->target_list;
            $$->var_list = $1->var_list;
            $$->prf = (char *) malloc(14 + strlen($1->prf) +
                                    strlen(pl->prf));
            sprintf($$->prf, "%s%s", $1->prf, pl->prf);

            free($1->prf); free($1);
            free(pl->prf); free(pl);
            printf("var list:>%s\n---\n", $$->var_list);
            printf("target_list:>%s\n------\n", $$->target_list);
            printf("prf:>%s\n-----\n", $$->prf);
      }
      ;
attribute :       IDENT DOT IDENT
      {
            $$ = getattype(stv($1), stv($3));
      }
      ;

string.value :    STRING
      { $$ = strsave(stv($1)); }
      ;
%%
#include "lex.c"

yyerror(s)
    char *s;
{
    printf("ERROR with yytext = :%s:\n", yytext);
}
main(argc,  argv)
      int argc;
      char *argv[];
{
    char *sf;
    FILE *fopen(), *freopen();

      if ((argc < 3) || (argc > 3)) {
            fprintf(stderr,
                  "Usage: impbif server::database filename\n");
            exit();
      }

    fp = fopen("./encoder.qc", "w");
    dbname = argv[1];
```

```
    freopen(argv[2], "r", stdin);
    eqlopen(argv[1], "-c10");

    yyparse();
    eqlclose();
}
struct  PARM_LIST  *mkattr(ad,  n)
      struct ATTRDEFN *ad;
      int n;
{
      struct PARM_LIST *pl;

      pl = (struct PARM_LIST *) malloc( sizeof(struct PARM_LIST));
      pl->target_list = retatctrl(ad);
      switch(ad->typ) {
            case 'c' :
                  pl->prf = (char *) malloc(26);
                  pl->var_list = (char *) malloc(20);
                  sprintf(pl->prf,"\tprintf(\"%%s\",svar);\n");
                  sprintf(pl->var_list, "\targv[%d] = svar;\n",n);
                  break;
            case 'i' :
                  pl->prf = (char *) malloc(26);
                  sprintf(pl->prf,"\tprintf(\"%%d\", ivar);\n");
                  pl->var_list = (char *) malloc(28);
                  sprintf(pl->var_list,
                        "\targv[%d] = (char *)&ivar;\n",n);
                  break;
            case 'f' :
                  if (ad->sz == 4) {
                        pl->prf = (char *) malloc(26);
                        sprintf(pl->prf,
                              "\tprintf(\"%%f\", fvar);\n");
                        pl->var_list = (char *) malloc(28);
                        sprintf(pl->var_list,
                              "\targv[%d] = (char *)&fvar;\n",n);
                  }
                  else {
                        pl->prf = (char *) malloc(27);
                        sprintf(pl->prf,
                              "\tprintf(\"%%lf\", dvar);\n");
                        pl->var_list = (char *) malloc(28);
                        sprintf(pl->var_list,
                              "\targv[%d] = (char *)&dvar;\n",n);
                  }
                  break;
            default : ;
      };
      return(pl);
}
struct  PARM_LIST  *mkstrprm(s)
      char *s;
{
      struct PARM_LIST *pl;

      pl = (struct PARM_LIST *) malloc( sizeof(struct PARM_LIST));
      pl->prf = (char *) malloc( strlen(s) + 13);
      sprintf(pl->prf, "\tprintf(\"%s\");\n", s);
      pl->var_list = strsave("");
      pl->target_list = strsave("");
      return(pl);
}
```

F.2 Lex definition File

```
%o4000
%{

#define input() ((!(rgcnt[rgflg]) && rgflg) ? (rgflg--, '@') :
(((yytchar=yysptr>yysbuf?U(*--
yysptr):getc(yyin))==10?(yylineno++,yytchar):yytchar)==EOF?0:yytchar))

%}

D           [0-9]
E           [E][-+]{D}{D}
W           [\t ]
CHAR        [A-Za-z]

%%
{W}*\n      ;

\;      { return( SEMI ); }
\:      { return( COLON ); }
\.      { return( DOT ); }
\{      { return( L_BRCK ); }

\}      { return( R_BRCK ); }

\:      { return( COLON ); }
{W}*[+-]?{D}+"."{D}*({E})?{W}*      |
{W}*[+-]?{D}*"."{D}+({E})?{W}*  {
                        newtkn( REAL );
                    dtv(yylval.t) = atof(yytext);
                    return( REAL );
                }
{W}*[+-]?{D}+{W}*       {
                            newtkn( INTEGER );
                    itv(yylval.t) = atoi(yytext);
                    return( INTEGER );
                }
\"({CHAR}*{D}*[\\\-\,!@#%^&()]*{W}*)*\"  {
    newtkn( STRING );
    stv(yylval.t) = strsave(yytext+1);
    stv(yylval.t)[yyleng-2] = '\0';
    return( STRING );
}
({CHAR}{D}*)+                   {
    newtkn( IDENT );
    stv(yylval.t) = strsave(yytext);
    return( IDENT );
}

@       { return( EORG ); }
%%
/*-------------------------------------------------------------*/
```

Appendix G

An Example of a Format Encoder

This appendix contains the listing of a format encoder for the character form of the BIF format.

G.1 tre.h

```
struct value {
      int type;
      union {
            int ival;
            double dval;
            char *sval;
      } value;
};
struct tnode {
      struct value *av[10];
      int (*prntfn)(), (*mksubtree)();
      struct tnode *sib;
      struct tnode *chld;
};
#define INT 50
#define REAL 51
#define CHAR 52
extern struct tnode *nwnd();
extern int prnt_tree();
```

G.2 tre.c

```
#include <stdio.h>
#include "tre.h"
struct  tnode  *nwnd()
{
      struct tnode *n;

      n = (struct tnode *) malloc( sizeof(*n));
      n->sib = NULL;n->chld = NULL;
      return(n);
}
incomplete_tree(hd)
        struct tnode *hd;
{
        int inc;
        inc = 0;
        if (hd != NULL)
            inc =
              ((hd->mksubtree != empty) || incomplete_tree(hd->chld)
                  || incomplete_tree(hd->sib));
        return(inc);
}
```

```
build_tree(hd)
        struct tnode *hd;
{
      if (hd != NULL) {
                build_tree(hd->chld);
                (*hd->mksubtree)(hd);
                hd->mksubtree = empty;
                build_tree(hd->sib);
        }
}
empty(hd)
        struct tnode *hd;
{}
int  file_gen(hd)
        struct tnode *hd;
{
        if (hd != NULL) {
                (*(hd->prntfn)) (hd->av);
                file_gen(hd->chld);
                file_gen(hd->sib);
        }
}
```

G.3 enc.qc

```
#include <stdio.h>
#include "tre.h"

#define POINT 0
#define CURVE 1
#define CIRCLE 2
#define TEXT 3

int prntdrw(), object(), crv_prm(), ln_seg(), arc_seg(), hook(),
      empty(), prmtree(), nametree(), objecttree(), crvetree();

##      int sq, obj, prm, prmt;
##      double xx, yy, blg, r, du, dcx, dcy, dsz;
##      char *styl, *oc, *ds, *o1, *o2;

struct tnode *init_tree();

main( argc, argv)
        int argc;
##      char *argv[];
{
        struct tnode *tree, *t;

        styl = (char *) malloc(30);
        oc = (char *)malloc(11); ds = (char *)malloc(16);
        o1 = (char *)malloc(16); o2 = (char *)malloc(16);

##      ingres argv[1]

        tree = init_tree();
        while (incomplete_tree(tree))
                build_tree(tree);

        file_gen(tree);
}
```

```
struct  tnode  *init_tree()
{
        struct tnode *hd;

        hd = nwnd();
        hd->mksubtree = objecttree;
        hd->prntfn = prntdrw;

        return(hd);
}

objecttree(hd)
        struct tnode *hd;
{
        struct tnode *t;

        t = NULL;

##      retrieve(obj = cmpobjnm.cmpobjid)
##{
        if (t == NULL)
                t = hd->chld = nwnd();
        else {
                t->sib = nwnd(); t = t->sib;
                }
        t->av[0] =  (struct value *) malloc(sizeof (*(t->av[0])));
        t->av[0]->type = INT;
        t->av[0]->value.ival = obj;
        t->prntfn = object;
        t->mksubtree = prmtree;
##}
}

object(av)
        struct value *av[];
{
        obj = av[0]->value.ival;
##      retrieve (oc = cmpobjnm.ocd, ds = cmpobjnm.dpso,
##                o1 = cmpobjnm.othr1, o2 = cmpobjnm.othr2,
##                xx = cmpobjnm.hookx, yy = cmpobjnm.hooky)
##         where cmpobjnm.cmpobjid= obj
        printf("name %s\nhook %lf %lf\n", oc, xx, yy);
}

prmtree(hd)
        struct tnode *hd;
{
        struct tnode *t;

        t = NULL;

        obj = hd->av[0]->value.ival;

##      retrieve(prm = cmpobjdfn.objectid, prmt =
cmpobjdfn.objecttype)
##         where cmpobjdfn.cmpobjid = obj
##{
        if (t == NULL)
                t = hd->chld = nwnd();
        else { t->sib = nwnd(); t = t->sib;
                }
```

```
        t->av[0] =  (struct value *) malloc(sizeof (*(t->av[0])));
        t->av[0]->type = INT;
        t->av[0]->value.ival = obj;
        t->av[1] =  (struct value *) malloc(sizeof (*(t->av[0])));
        t->av[1]->type = INT;
        t->av[1]->value.ival = prm;
        if (prmt == CURVE) {
                t->prntfn = crv_prm;
                t->mksubtree = crvetree;
        }
        else  { /** Kludge -- incomplete **/
                t->prntfn = empty;
                t->mksubtree = empty;
        }
##}
}
crvetree(hd)
        struct tnode *hd;
{
        struct tnode *t;

        t = NULL;
        prm = hd->av[1]->value.ival;
/** retrieve should get tuples in order of ascending seq **/

##      retrieve (blg = curvedfn.bulge, sq= curvedfn.seq)
##         where curvedfn.curveid = prm and curvedfn.seq != 0
##{
        if (t == NULL)
                t = hd->chld = nwnd();
        else {
                t->sib = nwnd(); t = t->sib;
                }
        if (blg == 0.0) {
                t->prntfn = ln_seg;
                t->mksubtree = empty;
        }
        else {
                t->prntfn = arc_seg;
                t->mksubtree = empty;
        }
        t->av[0] =  (struct value *) malloc(sizeof (*(t->av[0])));
        t->av[0]->type = INT;
        t->av[0]->value.ival = prm;
        t->av[1] =  (struct value *) malloc(sizeof (*(t->av[0])));
        t->av[1]->type = INT;
        t->av[1]->value.ival = sq;
##}
}
arc_seg(av)
        struct value *av[];
{
        prm = av[0]->value.ival;
        sq = av[1]->value.ival;

##      retrieve (xx = curvedfn.x, yy = curvedfn.y,
##                blg = curvedfn.bulge)
##         where curvedfn.curveid = prm and curvedfn.seq = sq
        printf("arc %lf %lf %lf\n", xx, yy, blg);
}
```

```
ln_seg(av)
        struct value *av[];
{
        prm = av[0]->value.ival;
        sq = av[1]->value.ival;

##      retrieve (xx = curvedfn.x, yy = curvedfn.y)
##         where curvedfn.curveid = prm and curvedfn.seq = sq
        printf("to %lf %lf\n", xx, yy);
}

crv_prm(av)
        struct value *av[];
{
        obj = av[0]->value.ival;
        prm = av[1]->value.ival;
##      retrieve (styl = cmpobjdfn.style)
##         where cmpobjdfn.objectid = obj
##      retrieve (xx = curvedfn.x, yy = curvedfn.y)
##         where curvedfn.curveid = prm and curvedfn.seq = 0
        printf("line %lf %lf %s\n",xx, yy, styl);
}
prntdrw(av)
      struct value *av[];
{
##    retrieve (du = drw.dunit, dcx = drw.dcenx, dcy = drw.dceny,
##              dsz = drw.size)
      printf("drawing %lf %lf %lf %lf\n", du, dcx, dcy, dsz);
```